intel Software

THE PARALLEL UNIVERSE

Heterogeneous Programming Using oneAPI

Accelerating Compression on Intel® FPGAs

Is Your Game GPU-Bound?

New Threading Capabilities in Julia v1.3

Issue 39 2020

CONTENTS

Sign up for future issues

LETTER FROM THE EDITOR

Henry A. Gabb, Senior Principal Engineer at Intel Corporation, is a longtime high-performance and parallel computing practitioner who has published numerous articles on parallel programming. He was editor/coauthor of "Developing Multithreaded Applications: A Platform Consistent Approach" and program manager of the Intel/Microsoft Universal Parallel Computing Research Centers.

Happy New Year

Welcome to the Era of oneAPI

Welcome to another year of *The Parallel Universe*. 2020 promises to be interesting. Some readers may know that I used to dread the trend toward heterogeneous computing. Then I came to accept it as inevitable. Now, I'm embracing it. Sure, heterogeneity is both a blessing and a curse. The blessing is better performance and efficiency. The curse is increased complexity. My hope is that someday machines will just take care of it for me (see **Why More Software Development Needs to Go to the Machines**), but until then, practical steps are being taken to minimize this complexity, starting with **oneAPI**. oneAPI is an **open specification** that describes a single software abstraction across diverse compute architectures.

The Intel implementation of oneAPI was recently announced by Raja Koduri (senior vice president, chief architect, and general manager of Intel Architecture, Graphics, and Software) at the **Intel® HPC Developer Conference**. Our feature article, **Heterogeneous Programming Using oneAPI**, gives an overview of this unified, standards-based approach to heterogeneous computing. **Accelerating Compression on Intel® FPGAs** shows how **Data Parallel C++** makes FPGAs more accessible. Continuing this theme of heterogeneity, **Is Your Game GPU-Bound?** shows you how to answer this question using analysis tools like **Intel® Graphics Performance Analyzers**.

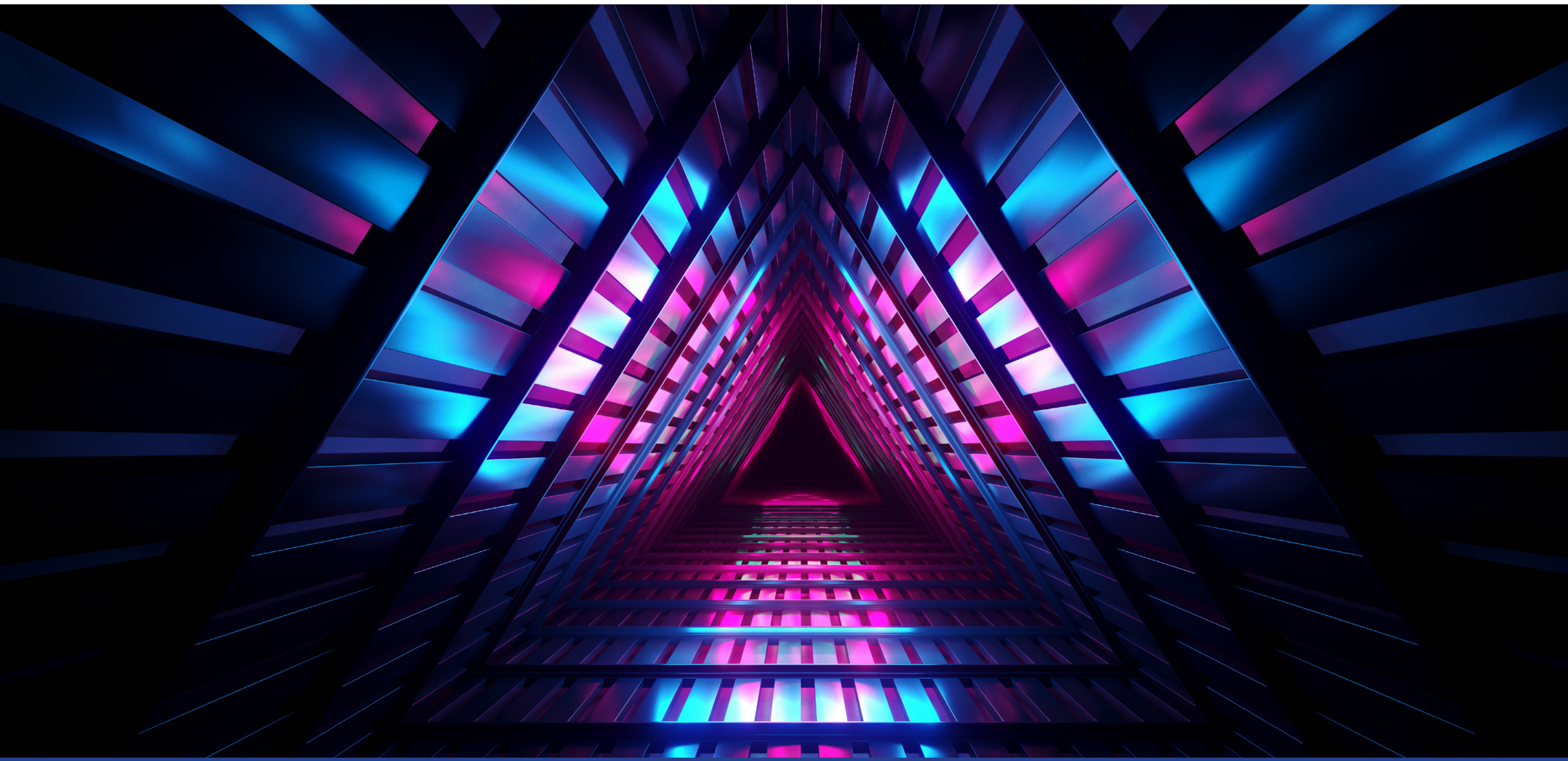
In the last issue, I briefly covered composable threading in the Julia* programming language. Jameson Nash and Jeff Bezanson from Julia Computing, Inc., and Kiran Pamnany from Caltech, were kind enough to provide a more detailed look at the **New Threading Capabilities in Julia v1.3** for this issue. They walk through several code examples illustrating task parallelism using Julia.

We close this issue with three articles on data analytics. The first, **Fast Gradient Boosting Tree Inference for Intel® Xeon® Processors**, shows how to use the XGBoost* library to improve the performance of model predictions. If you recall, the feature article in our **last issue** covered performance improvements for XGBoost training. The second, **K-means Acceleration with 2nd Generation Intel® Xeon® Scalable Processors**, shows how to take advantage of optimizations in **Intel® Distribution for Python*** and the **Intel® Data Analytics Acceleration Library** to do k-means clustering. Finally, in **Measuring Graph Analytics Performance**, I discuss the right ways—and wrong ways—to do graph analytics benchmarking. However, the graph analytics landscape is large and varied, so please let me know if you disagree with my assertions.

Expect to see more articles on oneAPI in future issues. And, as always, don't forget to check out **Tech.Decoded** for more information on Intel's solutions for code modernization, visual computing, data center and cloud computing, data science, and systems and IoT development.

Henry A. Gabb

January 2020



HETEROGENOUS PROGRAMMING USING ONEAPI

How to Deliver Uncompromised Performance for Diverse Workloads Across Multiple Architectures

Nitya Hariharan, Application Engineer; Rama Kishan Malladi, Performance Modeling Engineer; Amarpal S. Kapoor, Technical Consulting Engineer; Kevin P O'Leary, Technical Consulting Engineer; Intel Corporation

Getting the maximum achievable performance out of today's hardware is a fine balance between optimal use of underlying hardware features and using code that is portable, easily maintainable, and power-efficient. These factors don't necessarily work in tandem. They require prioritizing based on user needs. It's non-trivial for users to maintain separate code bases for different architectures. A standard, simplified programming model that can run seamlessly on scalar, vector, matrix, and spatial architectures will give developers greater productivity through increased code reuse and reduced training investment.

oneAPI is an industry initiative designed to deliver these benefits. It's based on standards and open specifications and includes the **Data Parallel C++ (DPC++)** language as well as a set of domain libraries. The goal of oneAPI is for hardware vendors across the industry to develop their own compatible implementations targeting their CPUs and accelerators. That way, developers only need to code in a single language and set of library APIs across multiple architectures and multiple vendor devices.

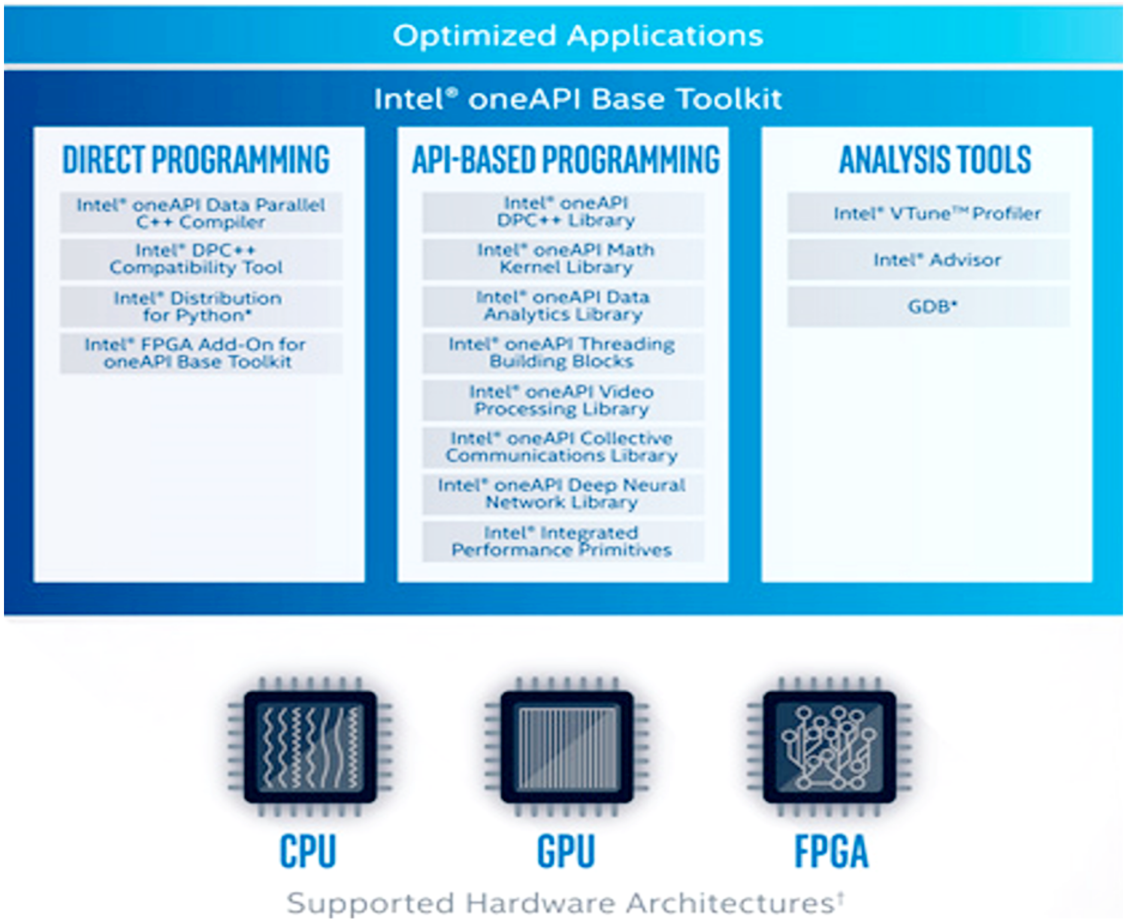
The Intel beta oneAPI developer tools implementation, targeting Intel® CPUs and accelerators, consists of the **Intel® oneAPI Base Toolkit** along with multiple domain specific toolkits—Intel® **HPC**, **IoT**, **DL Framework Developer**, and **Rendering** toolkits—which cater to different users.

Figure 1 shows the different layers that are part of the beta **Intel oneAPI** product and the Base Toolkit, which consists of the Intel oneAPI DPC++ Compiler, the Intel® DPC++ Compatibility Tool, multiple optimized libraries, and advanced analysis and debugging tools. Parallelism across architectures is expressed using the DPC++ language, which is based on SYCL* from Khronos Group. It uses modern C++ features along with Intel-specific extensions for efficient architecture usage. DPC++ language features allow code to be run on the CPU and to be offloaded onto an available accelerator—making it possibled to reuse code. A fallback property allows the code to be run on the CPU when an accelerator isn't available. The execution on the host and accelerator, along with the memory dependencies, are clearly defined.

Users can also port their codes from CUDA* to DPC++ using the Intel DPC++ Compatibility Tool. It assists developers with a one-time migration and typically migrates 80 to 90% of the code automatically.

In addition to DPC++, the Intel oneAPI HPC Toolkit supports OpenMP* 5.0 features that allow code to be offloaded onto a GPU. Users can either transition to using DPC++ or make use of the offload features on their existing C/C++/Fortran code. API-based programming is supported through a set of libraries (e.g., the **Intel® oneAPI Math Kernel Library**), which will be optimized for Intel GPUs.

The beta Intel oneAPI product also offers new features in **Intel® VTune™ Profiler**[1] and **Intel® Advisor**[2], which allow users to debug their code and look at performance-related metrics when code is offloaded onto an accelerator.


Optimized Applications
Intel® oneAPI Base Toolkit
DIRECT PROGRAMMING
Intel® oneAPI Data Parallel C++ Compiler
Intel® DPC++ Compatibility Tool
Intel® Distribution for Python*
Intel® FPGA Add-On for oneAPI Base Toolkit
API-BASED PROGRAMMING
Intel® oneAPI DPC++ Library
Intel® oneAPI Math Kernel Library
Intel® oneAPI Data Analytics Library
Intel® oneAPI Threading Building Blocks
Intel® oneAPI Video Processing Library
Intel® oneAPI Collective Communications Library
Intel® oneAPI Deep Neural Network Library
Intel® Integrated Performance Primitives
ANALYSIS TOOLS
Intel® VTune™ Profiler
Intel® Advisor
GDB*
CPU
GPU
FPGA
Supported Hardware Architectures†

1 **Components of the Beta Intel® oneAPI Base toolkit**

This article introduces the beta release of the oneAPI product to facilitate heterogeneous programming. We'll introduce the oneAPI software model and then discuss the compilation model and the binary generation procedure. oneAPI provides a single binary for all architectures, so the compile and link steps are different from normal methods of binary generation. Finally, we'll examine some sample programs. Note that we use the terms accelerator, target, and device interchangeably throughout this article.

oneAPI Software Model

The oneAPI software model, based on the SYCL specification, describes the interaction between the host and device in terms of code execution and memory usage. The model has four parts:

1. **A platform model** specifying the host and device
2. **An execution model** specifying the command queues and the commands that will be run on the device
3. **A memory model** specifying memory usage between the host and device
4. **A kernel model** that targets computational kernels to devices

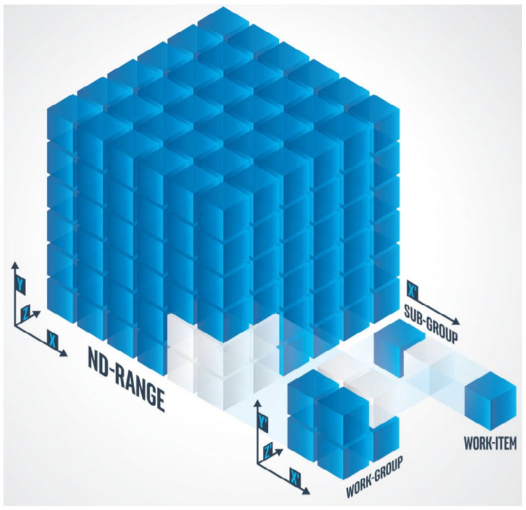
Platform Model

The oneAPI platform model specifies the host and multiple devices that communicate with each other or the host. The host controls the execution of kernels on the devices and coordinates among them if there are multiple devices. Each device can have multiple compute units. And each compute unit can have multiple processing elements. The oneAPI specification can support multiple devices like GPUs, FPGAs, and ASICs as long as the platform satisfies the minimum requirements of the oneAPI software model. This typically means the hosts need to have a specific operating system, a specific GNU* GCC version, and certain drivers needed by the devices. (See the release notes for each oneAPI component for details on the platform requirements.)

Execution Model

The oneAPI execution model specifies how the code is executed on the host and device. The host execution model creates command groups to coordinate the execution of kernels and data management between host and devices. The command groups are submitted in queues that can be run with either an in-order or out-of-order policy. Commands within a queue can be synchronized to ensure data updates on the device are available to the host before the next command is executed.

The device execution model specifies how the computation is done on the accelerator. The execution ranges over a set of elements that can either be a one-dimensional or multi-dimensional data set. This range is split into a hierarchy of ND-Range, work-groups, sub-groups, and work-items as shown in **Figure 2** for a three-dimensional case.



2 **Relationship between ND-Range, work-groups, sub-groups and work-items**

Note that this is similar to the SYCL model, with the exception of sub-groups, which are an Intel extension. The work-item is the smallest execution unit in the kernel. And the work-groups determine how data is shared among these work-items. These hierarchical layouts also determine the kind of memory that should be used to get better performance. For example, work-items typically operate on temporary data that's stored in the device memory and work-groups use global memory. The sub-group classification was introduced to provide support for hardware resources that have a vector unit. This allows parallel execution on elements.

From **Figure 2**, it's clear that the location of the work-group or work-item within ND-Range is important, since this determines the data point being updated within the computational kernel. The index into ND-Range that each work-item acts upon is determined using intrinsic functions in the `nd_item` class (`global_id, work_group_id`, `sub_group_id`, and `local_id`).

Memory Model

The oneAPI memory model defines the handling of memory objects by the host and device. It helps a user decide where memory will be allocated depending on the application's needs. Memory objects are classified as type buffer or images. An accessor can be used to indicate the location of the memory object and the mode of access. The accessor provides different access targets for objects residing on the host, global memory on the device, the device's local memory, or images residing on the host. The access types can be read, write, atomic, or read and write.

The Unified Shared Memory model allows the host and device to share memory without the use of explicit accessors. Synchronization using events manages the dependencies between host and device. A user can either explicitly specify an event to control when data updated by a host or device is available for reuse, or implicitly depend on the runtime and device drivers to determine this.

Kernel Programming Model

The oneAPI kernel programming model specifies the code that's executed on the host and device. Parallelism isn't automatic. The user needs to specify it explicitly using language constructs.

The DPC++ language requires a compiler that can support C++11 and later features on the host side. The device code, however, requires a compiler that supports C++03 features and certain C++11 features like lambda expressions, variadic templates, rvalue references, and alias templates. It also requires `std::string`, `std::vector`, and `std::function` support. There are restrictions on certain features for the device code which include virtual functions and virtual inheritance, exception handling, run-time type information (RTTI), and object management employing new and delete operators.

The user can decide to use different schemes to describe the separation between the host and device code. A lambda expression can keep the kernel code in line with the host code. A functor keeps the host code in the same source file, but in a separate function. For users who are porting OpenCL code, or those who require an explicit interface between the host and device code, the kernel class provides the necessary interface.

The user can implement parallelism in three different ways:

- **A single task** that executes the whole kernel in a single work-item
- **The `parallel_for` construct**, which distributes the tasks among the processing elements
- **The `parallel_for_work_group`**. The `parallel_for_work_group` construct distributes the tasks among the work-groups and can synchronize work-items within a work-group through the use of barriers.

oneAPI Compilation Model

The oneAPI compilation model consists of build and link steps. However, the binary generated needs to support the execution of the device code on multiple accelerators. This means a DPC++ compiler and linker have to carry out additional commands to generate the binary. This complexity is generally hidden from the user, but can be useful for generating target-specific binaries.

The host code compilation is done in the default way for a standard x86 architecture. The binary generation for the accelerator is more complex because it needs to support single or multiple accelerators in addition to optimizations that are specific to each accelerator. This accelerator binary, known as a fat binary, contains a combination of:

- **An intermediate Standard Portable Intermediate Representation (SPIR-V) representation**, which is device-independent and generates a device-specific binary during compilation.
- **Target-specific binaries** that are generated at compile-time. Since oneAPI is meant to support multiple accelerators, multiple code forms are created.

Multiple tools generate these code representations, including the clang driver, the host and device DPC++ compiler, the standard Linux* (`ld`) or Windows* (`link.exe`) linker, and tools to generate the fat object file. During execution, the oneAPI runtime environment checks for a device-specific image within the fat binary and executes it, if available. Otherwise, the SPIR-V image is used to generate the target-specific image.

oneAPI Programming Examples

In this section, we look at sample code for the beta Intel oneAPI DPC++ Compiler, OpenMP device offload, and the Intel DPC++ Compatibility Tool.

Writing DPC++ Code

Writing DPC++ code requires a user to exploit the APIs and syntax of the language. **Listing 1** shows some sample code conversion from C++ (CPU) code to a DPC++ (host and accelerator) code. It's an implementation of the Högbom CLEAN* algorithm posted on GitHub[4]. The algorithm iteratively finds the highest value in the image and subtracts a small gain of this point source convolved with the point spread function of the observation until the highest value is smaller than some threshold. The implementation has two functions: `findPeak` and `subtractPSF`. These have to be ported from C++ to DPC++ as shown in **Listings 1** and **2**.

Listing 1. Baseline and DPC++ implementation of the `subtractPSF` code

```
subtractPSF C++ code: baseline
for (int y = starty; y <= stopy; ++y) {
  lhsIdx = y * residualWidth + startx;
  rhsIdx = (y - diffy) * psfWidth + (startx - diffx);
  for (int x = startx; x <= stopx; ++x, lhsIdx++, rhsIdx++)
    residual[lhsIdx] -= MUL * psf[rhsIdx];
}
```

```
subtractPSF DPC++ code
sycl::gpu_selector device_selector;
sycl::queue d_queue(device_selector);
sycl::buffer<float, 1>  a_device(psf.data(), psf.size());
sycl::buffer<float, 1>  b_device(residual.data(), residual.size());
unsigned long size_par = (stopy - starty);
sycl::range<1> a_size{size_par};
auto offset = sycl::id<1>(starty);

d_queue.submit([&](sycl::handler &cgh) {
  auto a_in  = a_device.get_access<sycl::access::mode::read>(cgh);
  auto b_res = b_device.get_access<sycl::access::mode::write>(cgh);
  cgh.parallel_for<class T>(a_size,offset,[=](sycl::id<1> idx) {
    int y = idx[0];
    int lhsIdx = y * residualWidth + startx;
    int rhsIdx = (y - diffy) * psfWidth + (startx - diffx);
    for (int x = startx; x < stopx; ++x, lhsIdx++, rhsIdx++) {
      b_res[lhsIdx] -= MUL * a_in[rhsIdx];
    }
  });
});
d_queue.wait_and_throw();
```

Code changes required to port from C/C++ to DPC++ include:

- **Introduction of the device queue** for a given device (using the device selector API)
- **Buffers created/accessed on the devic**e (using the `sycl::buffer/get_access` APIs)
- **Invocation of the `parallel_for`** to spawn/execute the computational kernel
- **Wait for the completion of the kernel execution** (and optionally catch any exceptions)
- **Intel® DPC++ Compiler and flags:** `dpcpp -std=c++11 -O2 -lsycl -lOpenCL`

Listing 2 shows the code changes for the `findPeak` function implementation. To better exploit parallelism in the hardware, DPC++ code has support for `local_work_size`, `global_id/local_id`, `workgroup`, and many other APIs, similar to the constructs used in OpenCL and OpenMP.

Listing 2. Baseline (top) and DPC++ (bottom) implementation of the `findPeak` code. `clPeak` is a structure of value and position data. Concurrent execution of work-groups is accomplished using the global and local IDs, and barrier synchronization across multiple threads (work-items) in a work-group. The result of this `parallel_for` execution is further reduced (not shown) to determine the maximum value and position across work-groups.

```
findPeak: Baseline
maxVal = 0.0;
maxPos = 0;
const size_t size = image.size();
for (size_t i = 0; i < size; ++i) {
  if (abs(image[i]) > abs(maxVal)) {
    maxVal = image[i];
    maxPos = i;
  }
}
```

```
findPeak: DPC++
auto bufacc = buf.get_access<sycl::access::mode::read>(cgh);
auto resacc = res.get_access<sycl::access::mode::read_write>(cgh);
sycl::accessor<clPeak, 1, sycl::access::mode::read_write,
               sycl::access::target::local>
               local_res(sycl::range<1>(local_size), cgh);
cgh.parallel_for<class ex1>
                (sycl::nd_range<1>(sycl::range<1>(global_size),
                 sycl::range<1>(local_size)),[=](sycl::nd_item<1> item)
{
  size_t global_id = item.get_global_id(0);
  size_t local_id  = item.get_local_id(0);
  size_t local_dim = item.get_local_range(0);
  size_t group_id  = item.get_group(0);

  local_res[local_id].val = 0.0;
  local_res[local_id].pos = 0;

  if(fabs(bufacc[global_id]) > fabs(local_res[local_id].val)){
    local_res[local_id].val = bufacc[global_id];
    local_res[local_id].pos = global_id;
  }

  item.barrier(sycl::access::fence_space::local_space);
  if (local_id == 0) {
    resacc[group_id].val = 0.0;
    resacc[group_id].pos = 0;
    for (int i=0; i  < local_dim; i++) {
      if(fabs(local_res[i].val) > fabs(resacc[group_id].val)){
        resacc[group_id].val = local_res[i].val;
        resacc[group_id].pos = local_res[i].pos;
      }
    }
  }
});
```

OpenMP Offload Support

The beta Intel oneAPI HPC Toolkit provides OpenMP offload support, which enables users to take advantage of OpenMP device offload features. We look at a sample open-source Jacobi code[3] written in C++ with OpenMP pragmas. The code has a main iteration step that:

- **Calculates** the Jacobi update
- **Calculates** the difference between the old and new solution
- **Updates** the old solution
- **Calculates** the residual

The iteration code snippet is shown in **Listing 3**.

Listing 3. Sample Jacobi solver with OpenMP pragmas

```
// Iterate M times
#pragma omp parallel private (i, t)
{
#pragma omp for { // Jacobi update on xnew using b and x }
#pragma omp for reduction (+:d) { // Calculate difference d }
#pragma omp for { //overwrite x with x_new }
#pragma omp for reduction (+:r) { // Calculate residual r }
}
```

Listing 4 shows the updated code with the `omp` target clause, which can be used to specify the data to be transferred to the device environment, along with a data modifier that can either be `to`, `from`, `tofrom`, or `alloc`. Since array `b` is not modified, we use the clause `to`. And since `x` and `xnew` are initialized before the offload directives and updated within the device environment, we use the `tofrom` clause. The reduction variables `d` and `r` are also set and updated during each iteration and have the `tofrom` map clause.

Listing 4. Sample Jacobi solver updated with OpenMP offload pragmas

```
// Iterate M times
#pragma omp target data map(tofrom:x,xnew) map(to:n,b) map(tofrom:d,r)
{
#pragma omp parallel for { // Jacobi update }
#pragma omp parallel for reduction (+:d) { // Calculate difference d }
#pragma omp parallel for { // Overwrite x_old with x_new }
#pragma omp parallel for reduction (+:r) { // Calculate residual r }
}
```

To compile the offload target code with the oneAPI compiler, the user needs to set:

- **The environment variables** pertaining to the compiler path
- **The relevant** libraries
- **The different** components

The path to these environment variables will depend on the oneAPI setup on the user machine. We look at the compilation process for now, which will be similar across machines, to demonstrate the ease of use of the specification. To compile the code, use the LLVM-based `icx` or `icpc -qnextgen` compiler as follows:

```
$ icpc -fiopenmp -fopenmp-targets=spir64 -D__STRICT_ANSI__ jacobi.cpp -o
jacobi
```

The `-D__STRICT__ANSI` flag ensures compatibility with GCC 7.x and higher systems. The `spir64` flag refers to the target independent representation of the code and is ported to target-specific code during the link stage or execution. To execute the code, run these commands:

```
$ export OMP_TARGET_OFFLOAD=”MANDATORY”
$ export LIBOMPTARGET_DEBUG=1
$./jacobi
```

The `MANDATORY` option for `OMP_TARGET_OFFLOAD` indicates that the offload has to be run on the GPU. It's set to `DEFAULT` by default, which indicates offload can be run on CPU and GPU. The `LIBOMPTARGET_DEBUG` flag, when set, provides offload runtime information that helps in debugging.

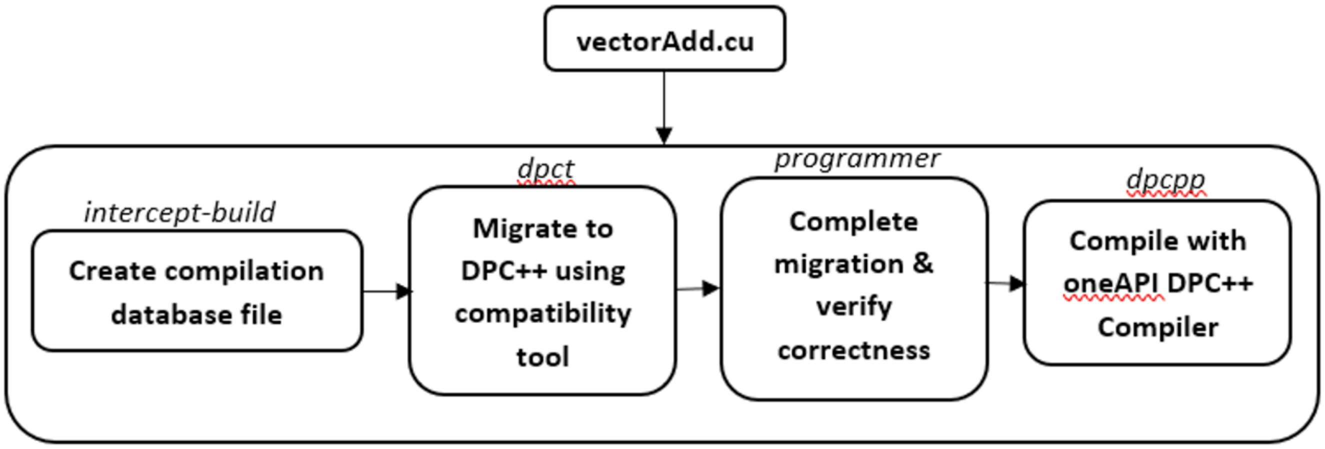
The OpenMP offload support example is for C/C++ programs, but Fortran offload is also supported. This allows HPC users with Fortran code bases to run their code on GPUs as well.

Intel DPC++ Compatibility Tool

The Intel DPC++ Compatibility Tool is a command-line-based code migration tool available as part of the Intel oneAPI Base Toolkit. Its primary role is to enable the porting of existing CUDA sources to DPC++. Source locations where automatic migration isn't possible are flagged through suitable errors and warnings. The Intel DPC++ Compatibility Tool also inserts comments in source locations where user interventions are necessary.

Figure 3 shows a typical workflow that CUDA users can use to port their source code to DPC++. The Intel DPC++ Compatibility Tool currently supports the Linux* and Windows* operating systems. This article assumes a Linux environment. The Intel DPC++ Compatibility Tool currently requires header files that are shipped with CUDA SDK. To demonstrate the migration process, we use the `VectorAdd` sample from CUDA SDK 10.1, typically found in a location similar to:

```
$ ls /usr/local/cuda-10.1/samples/0_Simple/vectorAdd
```


vectorAdd.cu
intercept-build
Create compilation database file
dpct
Migrate to DPC++ using compatibility tool
programmer
Complete migration & verify correctness
dpcpp
Compile with oneAPI DPC++ Compiler

3 **Recommended workflow for migrating existing CUDA applications**

VectorAdd is a single-source example with around 150 lines of code. The CUDA kernel device code in this case computes the vector addition of arrays A and B into array C.

Note that the commands, paths, and procedure shown here are correct at the time of publishing. Some changes may be introduced in the final version of the product.

To initialize the environment to use the Intel DPC++ Compatibility Tool, run the following command:

```
$ source /opt/intel/inteloneapi/setvars.sh
```

The `setvars.sh` script not only initializes the environment for the Intel DPC++ Compatibility Tool, but all other tools available in the Intel oneAPI Base Toolkit.

A simplified version of the CUDA Makefile is used, as shown in **Listing 5**.

LIsting 5. Makefile for porting CUDA code to DPC++

```
$ cat Makefile
CC=/usr/local/cuda-10.1/bin/nvcc
CCFLAGS=-m64 -gencode arch=compute_50,code=sm_50 -gencode
arch=compute_52,code=sm_52 -gencode arch=compute_60,code=sm_60 -gencode
arch=compute_61,code=sm_61 -gencode arch=compute_70,code=sm_70 -gencode
arch=compute_75,code=sm_75 -gencode arch=compute_75,code=compute_75
OBJECT=vectoradd.o
SOURCE=vectoradd.cu
main:
        $(CC) $(CCFLAGS) -o $(OBJECT) -c $(SOURCE)
clean:
        rm -rf vectoradd.o
```

The next step intercepts commands issued as the Makefile executes and stores them in a compilation database file in JSON format. The Intel DPC++ Compatibility Tool provides a utility called `intercept-build` for this purpose. Here's a sample invocation:

```
$ intercept-build make
```

The real conversion step is then invoked:

```
$ dpct -p compile_commands.json --in-root=. --out-root=dpct_output
vectorAdd.cu
```

The `--in-root` and `--out-root` flags set the location of user program source and location where the migrated DPC++ code must be written. This step generates `./dpct_output/vectorAdd.dp.cpp`.

To ensure that vector addition is deployed onto the integrated GPU, explicit specification of the GPU queue is made instead of the submission to the default queue. The list of supported platforms is obtained with the list of devices for each platform by calling `get_platforms()` and `platform.get_devices()`. With the target device identified, a queue is constructed for the integrated GPU and the vector add kernel is dispatched to this queue. Such a methodology may be used to target multiple independent kernels to different target devices connected to the same host/node.

Next, the modified DPC++ code is compiled using:

```
$ dpcpp -std=c++11 -I=/usr/local/cuda-10.1/samples/common/inc
vectorAdd.dp.cpp -lOpenCL
```

The resulting binary is then invoked, and the vector addition is confirmed to be executing on the integrated GPU, shown in **Listing 6**.

Listing 6. Output from running the ported DPC++ code on the integrated GPU

```
$ ./a.out
[Vector addition of 5000000 elements]
Platform: Intel(R) OpenCL HD Graphics
 Device: Intel(R) Gen9 HD Graphics NEO
Platform: Intel(R) OpenCL
 Device: Intel(R) Core(TM) i7-7567U CPU @ 3.50GHz
Platform: SYCL host platform
 Device: SYCL host device
Copy input data from the host memory to the device
kernel launch with 19532 blocks of 256 threads
Running on :Intel(R) Gen9 HD Graphics NEO
Copy output data from the device to the host memory
Test PASSED
```

For details on these tools, use these help flags:

```
$ intercept-build -h
$ dpct -h
```

Uncompromised Performance for Diverse Workloads Across Multiple Architectures

This article introduced oneAPI and the beta Intel oneAPI Toolkits and outlined the components that are part of the Intel oneAPI Base Toolkit. The beta release includes toolkits to help users in the HPC, AI, analytics, deep learning, IoT, and video analytics domains transition to oneAPI. The DPC++ programming guide provides complete details on the various constructs supported for optimized accelerator performance. The OpenMP example shown in the article is for a C++ program. However, GPU offload will be supported for C and Fortran as well. oneAPI provides the software ecosystem you need to port and run your code on multiple accelerators.

References

1. **Intel® VTune™ Profiler**
2. **Intel® Advisor**
3. **Jacobi solver using OpenMP**
4. **https://github.com/ATNF/askap-benchmarks/tree/master/tHogbomCleanOMP**

intel®
Software

INTEL® DEVCLOUD A DEVELOPMENT SANDBOX FOR DATA CENTER TO EDGE WORKLOADS

Develop, Test, and Run Your Workloads on a Cluster of the Latest Intel® Hardware and Software.

With integrated Intel® optimized frameworks, tools, and libraries, you'll have everything you need for your projects.

Try Out a Diverse Collection of Intel® Hardware.

Expand your skills and experiment with this state-of-the-art cluster that offers capabilities like natural language processing and time-series analysis—plus edge acceleration hardware.

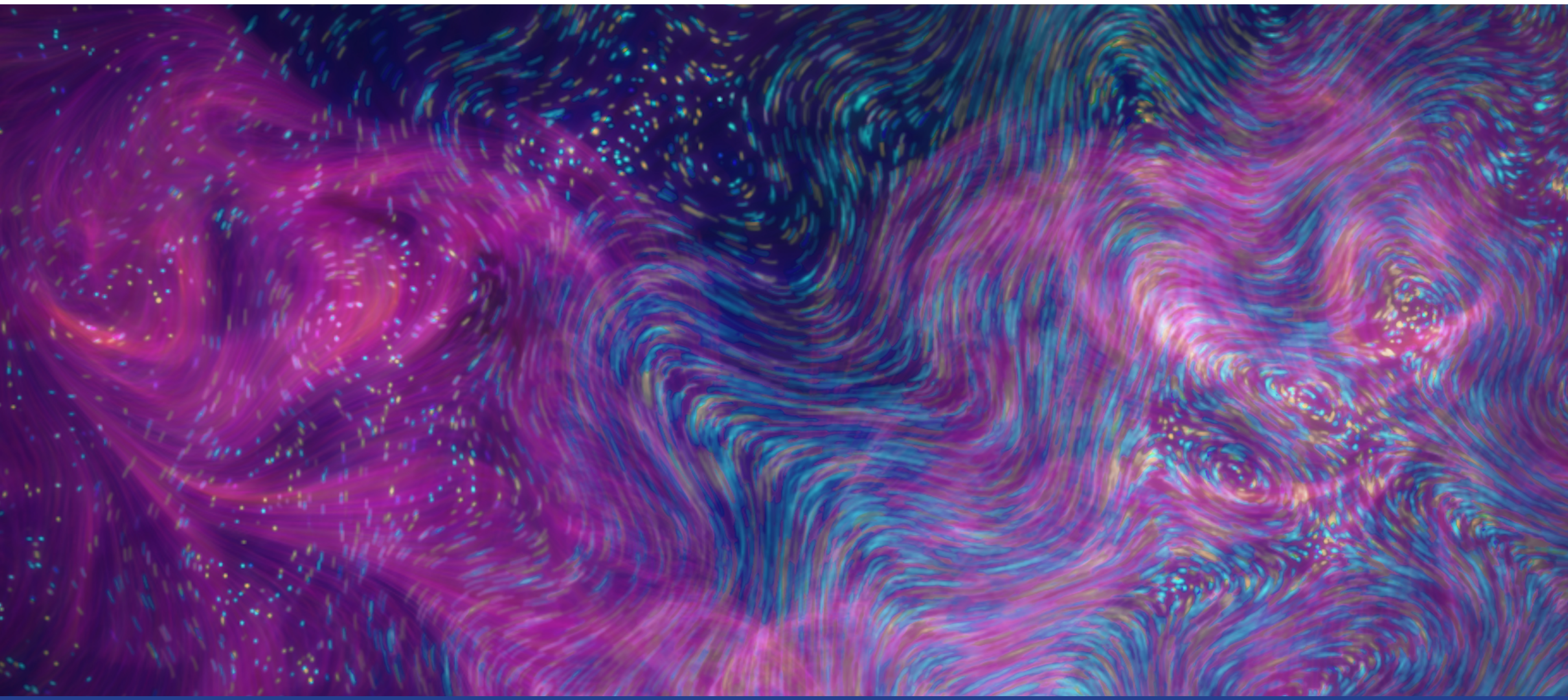
Develop with Intel® Software Development Tools.

Jump-start your projects without having to download, configure, or install the latest compilers, performance libraries, and tools from Intel.

Use Popular AI Frameworks.

Accelerate your algorithms and applications with Intel® Optimized AI Frameworks that are ready for training and inference.

GET STARTED NOW >



ACCELERATING COMPRESSION ON INTEL® FPGAS

How oneAPI is Making FPGAs More Accessible than Ever

Andrei Hagiescu, FPGA Software Engineer, and David Cashman, FPGA Software Engineer, Intel Corporation

Field programmable gate arrays (FPGAs) provide a flexible hardware platform that can achieve high performance on a large variety of workloads. In this article, we'll discuss the Intel GZIP example design, implemented with **oneAPI**, and how it can help make FPGAs more accessible. (For more on oneAPI, see this issue's feature article, **Heterogeneous Programming Using oneAPI**.)

The example design implements DEFLATE, a lossless data compression algorithm essential to many storage and networking applications. The example is written in **SYCL** and compiled using the **oneAPI Data Parallel C++ (DPC++) Compiler**, demonstrating a significant acceleration in compression times with compelling compression ratio. The GZIP example design takes advantage of the massive spatial parallelism available in FPGAs to accelerate the LZ77 compression algorithm by parallelizing memory accesses, dictionary searches, and matching. Since the design is implemented using oneAPI, the code can target any compute technology, but we specifically optimize for FPGAs. The example produces GZIP-compatible compressed data files so that developers can use standard software tools to decompress the compressed files produced by this design.

How FPGAs Work

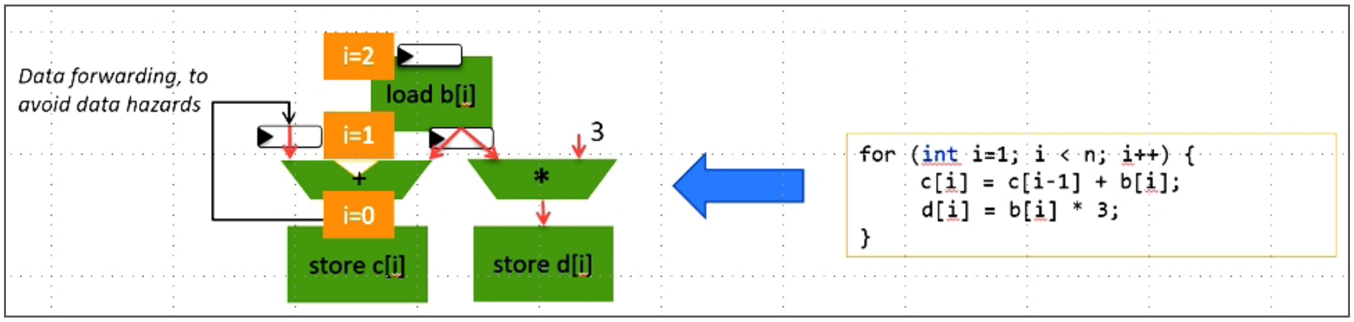
FPGAs are reconfigurable devices consisting of many low-level compute and storage elements (e.g., adders/multipliers, logic operations, memories) structured as a 2D array and connected by reconfigurable routing. These elements can form complex compute pipelines and specialized on-chip memory systems. In contrast to traditional architectures, which are designed to execute generic code, FPGAs can be reconfigured to implement custom architectures that boost the performance of a target application. For example, FPGAs can implement specialized compute pipelines that can execute an entire loop iteration every clock cycle. FPGAs fit well in an acceleration model, with off-chip attached memory (e.g., DDR4) and connectivity to a host CPU (e.g., through PCIe).

DPC++ Compilation to FPGAs

The FPGA backend of the DPC++ Compiler produces a bitstream that reconfigures the FPGA for the given code. Each kernel in the SYCL program is implemented using a subset of the FPGA resources. All implemented kernels can execute concurrently.

Within a kernel, each loop body is translated to a deep and specialized pipeline that contains all the functional units required to process an entire loop iteration. Conditional statements are refactored as predicated execution. Traversing the pipeline once executes an entire loop iteration.

The compiler identifies instructions that can be executed in parallel, and places them in the same pipeline stages, so that they execute in parallel on arbitrarily many functional units. Further, it optimizes the pipeline by ensuring that data can be forwarded, so that subsequent loop iterations can be issued as soon as possible (often in consecutive cycles), without data hazards (**Figure 1**).



```
for (int i=1; i < n; i++) {
    c[i] = c[i-1] + b[i];
    d[i] = b[i] * 3;
}
```

1 Pipeline generated from user code

Compiling for the FPGA typically takes several hours. To accelerate the development cycle, the FPGA backend is accompanied by an emulator, static performance reports, and a dynamic profiler to guide optimization decisions. Emulation and static report generation take minutes instead of hours, vastly improving the productivity of FPGA application development compared to traditional RTL development flows.

The oneAPI Advantage

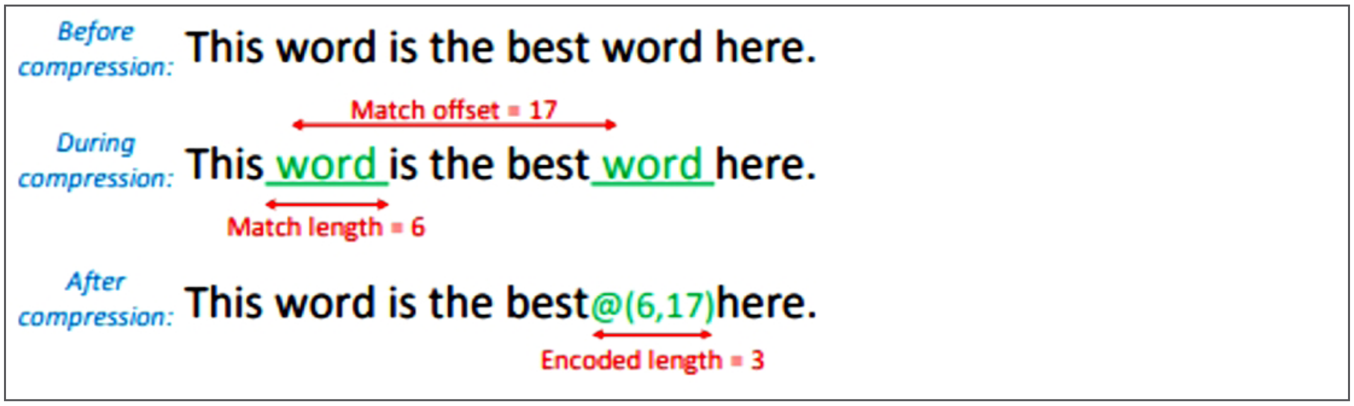
The GZIP design takes full advantage of key oneAPI programming features such as single-source design and multiple accelerators, as well as heterogeneous computing where the accelerator runs the hotspot code and the CPU is processing the non-hotspot code. Since the design is written using oneAPI, this enables anyone who knows C++ to write an algorithm suitable for compilation to FPGAs, focusing mainly on the algorithmic details and leaving the hardware translation to the compiler backend. Since the algorithm is expressed in C++, it can be easily tested for correct functionality. Hardware performance can be anticipated by examining reports before committing to a hardware compilation.

The Example Design

The GZIP design's architecture follows DEFLATE's dataflow. We create three kernels to do the work:

1. LZ77
2. Huffman
3. CRC

Our first kernel computes LZ77 data, searching and eliminating duplicate sequences from the file. The idea is illustrated in **Figure 2**.



2 **Example of LZ77 compression**

A duplicated sequence is replaced with a relative reference to the previous occurrence. Less storage is required to encode the reference than the original text, reducing the file size.

To find matches, we need to remember all the sequences we've seen and pick the best candidate match. The sequence is replaced, and the matching process continues from the end of the current match—or, if we didn't find a good match, at the next symbol. This is harder than it sounds.

Our goal will be to process 16 bytes on every FPGA clock cycle. So, in one clock cycle, we need to:

- **Look in our history** for the best match at each of 16 starting points
- **Pick the best match** (or matches, if there are several that don't overlap)
- **Write the result** to our output
- **Store the string** we just read back into the dictionary

It's not obvious that this work can be done in parallel, since matches can't overlap. We can't pick a match starting at a given byte until we know that no earlier match already covers it. (We'll cover this in detail in the next section.)

The second kernel applies Huffman encoding to the data, generating the final compressed result. During this step, all symbols and references are replaced with a variable bitwidth encoding which provides codes with fewer bits for the most frequent symbols, further reducing the file size. It's easier to see how this step can be parallelized: we can have 16 (or more) independent units of hardware, each determining the Huffman code for a given symbol. There's still a problem, though. Since the output has variable length, we need to eventually write each output to the right location in the output stream—which means we need to know how large all the previous outputs were. Also, Huffman codes are not byte-aligned, so we'll need to do a lot of bit-level manipulation. Fundamentally, though, this is a less complicated problem than LZ77, and we won't go into more detail here.

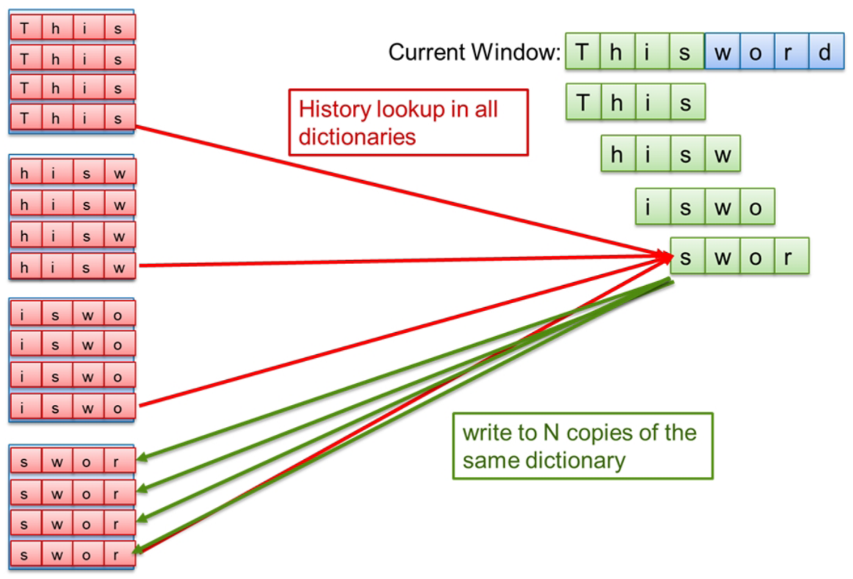
Finally, a third kernel computes CRC32 on the input data. This kernel is independent of the other two and can operate in parallel. It's also relatively simple to implement on the FPGA, so we won't discuss it in detail here.

FPGA-Optimized LZ77 Implementation

To capture the sequential nature of the data processing, the LZ77 encoder is described as a single task with a single main loop. That is, the code describes a single thread of execution, iterating symbol-by-symbol on the entire file. A set of dictionaries is created to store previously seen sequences in the datafile. These dictionaries are indexed through a hash of the data they store, similar to a hash map. However, for performance reasons, a more recent (colliding) entry overwrites dictionary data corresponding to an earlier entry with the same hash key. The dictionaries are updated by writing the newly seen input to all the dictionaries. To reduce the impact of collisions in the dictionaries, we separate the previously seen sequences in multiple disjoint sets.

Why do we need more than one dictionary? Recall that we want to process 16 bytes per cycle. This means storing 16 strings (one starting at each byte) to the dictionary and doing 16 hash lookups on every clock cycle. Each FPGA memory block only has two ports. To allow all these concurrent accesses, we create 16 separate memory systems, each storing strings at a different position. We now have our history spread across 16 dictionaries, so each of our 16 hash lookups now needs to be done in 16 different dictionaries—for a total of 256 lookups. It seems like we've just made the problem worse.

We can solve this by building 16 copies of each of our dictionaries, for a total of 256 dictionaries. All of these dictionaries use a large fraction of the FPGA's on-chip memory, but we've achieved our goal of doing 16 hash lookups and writes on every clock cycle. **Figure 3** shows an example of the dictionary structure we would need if we only wanted to process four bytes per clock cycle. (Incidentally, the FPGA area devoted to dictionaries is the main limitation preventing us from processing more than 16 bytes per cycle.)


Current Window: T h i s w o r d
History lookup in all dictionaries
write to N copies of the same dictionary

3 **Dictionary replication for four-byte parallel access**

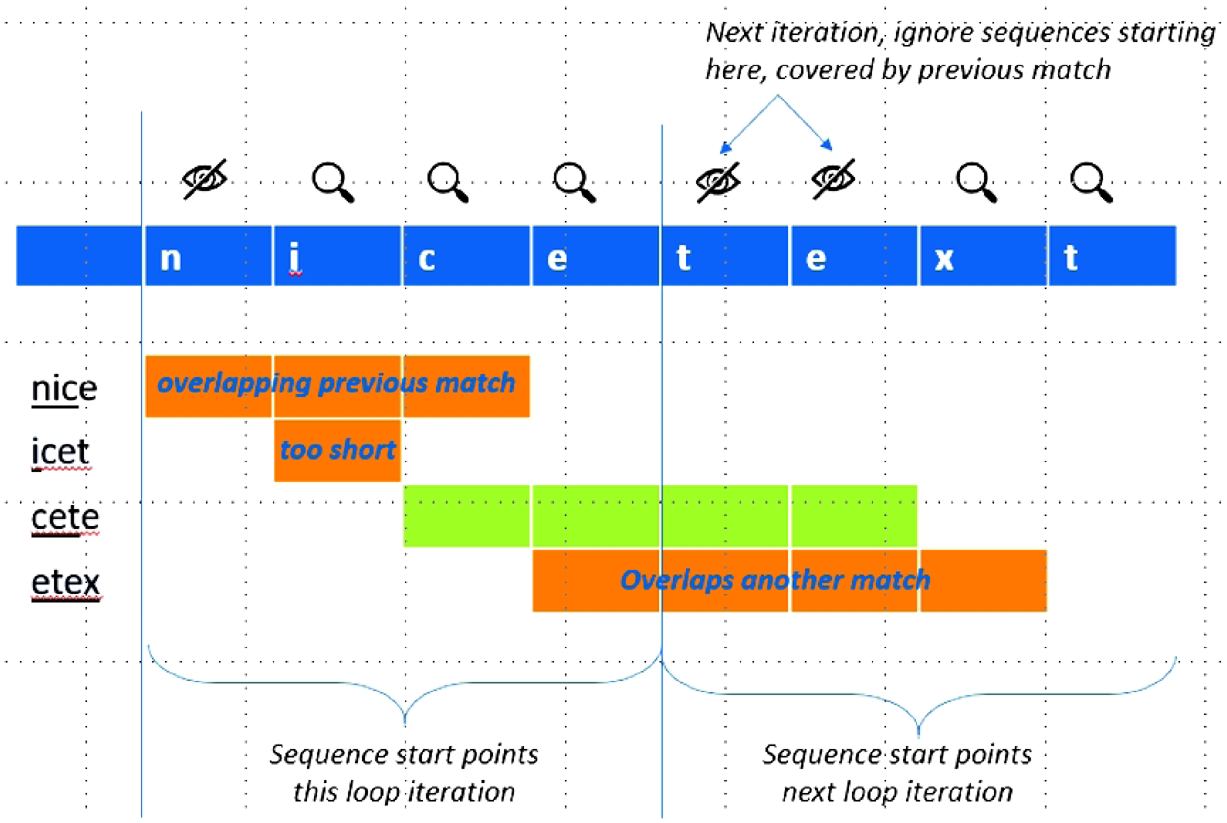
Expressing the parallel behavior may sound tricky. But, in fact, the compiler identifies the data parallelism between the lookups on its own. That is, the code will result in specialized FPGA logic capable of executing all the dictionary lookups concurrently. In the code, VEC is the number of bytes being processed per cycle, and LEN is the size of the string. In the example design, both are set to 16. We've used template metaprogramming (the unroller) to replicate the code in the inner function for all `i` and `j`. The data from all the lookups is aggregated in a reduction-like operation as part of the same pipeline.

Our dictionaries solve one problem, but we have a lot more processing to do if we want to complete a full loop iteration on each clock cycle. Luckily, we don't need to. The FPGA compiler will automatically pipeline the datapath. So after iteration 0 of the loop finishes reading from the dictionary, iteration 1 can start, while iteration 0 starts on picking the best match. The match selection can be staged across several cycles, with a different iteration of the loop operating at each stage.

```
//loop over VEC compare windows, each has a different hash
unroller<0, VEC>::step([&] (int i){
        //loop over all VEC bytes
        unroller<0, LEN>::step{[&] {int j) {

                compare_window[j][0][i] = dictionary_0[hash[1][j];
                compare_window[j][1][i] = dictionary_0[hash[1][j];
                compare_window[j][2][i] = dictionary_0[hash[1][j];
                compare_window[j][3][i] = dictionary_0[hash[1][j];
        }};
}};
```

When generating the output, one final challenge is how to correctly account for the impact of a match on subsequent matches. Once a match is identified, overlapping matches must be disregarded. In our `single_task` code representation, this manifests as a loop-carried variable that needs to be computed before subsequent iterations of the loop can proceed. If the computation is too complex, it may limit the clock speed of the FPGA, or limit our ability to complete a loop iteration on every clock cycle.

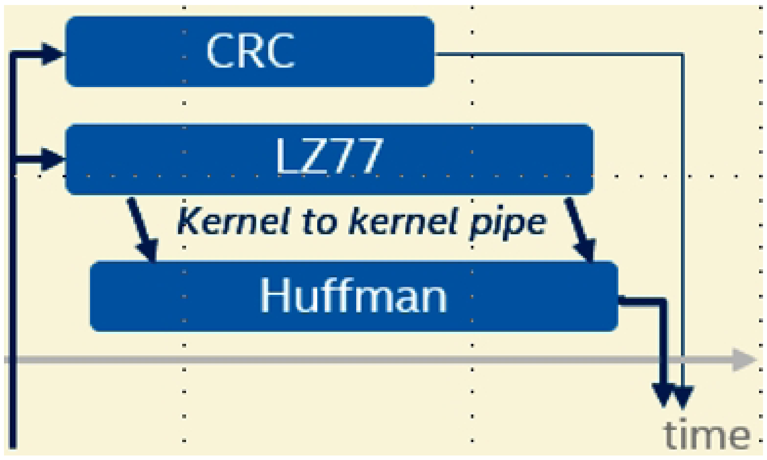
Because we target an FPGA, we can customize the hardware to optimize the handling of this dependency. We simplify the data hazard by minimizing the amount of computation that depends on it. For example, we choose to do the hash lookups speculatively, determining all the possible matches, even if the lookups are related to overlapped matches. We simply prune the overlapping matches later on. The FPGA backend optimizes the required forwarding logic, fully avoiding the data hazard. **Figure 4** demonstrates the pruning of speculative matches.


Next iteration, ignore sequences starting here, covered by previous match
n
i
c
e
t
e
x
t
nice
icet
cete
etex
overlapping previous match
too short
Overlaps another match
Sequence start points this loop iteration
Sequence start points next loop iteration

4 Pruning speculative matches, accounting for matches identified in previous loop iterations, overlapped matches in the current loop iteration, and poor quality matches

Task-Level Parallelism

To compress a file, the three processing kernels are asynchronously submitted by the host CPU. They can run concurrently in the FPGA hardware. They operate at slightly different rates, with CRC being faster. LZ77 produces data needed by the Huffman kernel. To avoid an additional delay, and to avoid transferring data through off-chip memory, we use kernel-to-kernel communication pipes, a proposed Intel extension of the SYCL language. This extension allows different kernels to exchange data, in sequence, without writing to off-chip memory, as shown in **Figure 5**.



5 **Task parallelism in GZIP**

Building the GZIP Design

You can **download and run the GZIP design** from the **oneAPI code samples**. You may target the FPGA emulation to verify correctness and functional behavior. Attempting to compress, for example, `/bin/emacs-24.3` will result in:

```
make fpga_emu
./gzip.fpga_emu /bin/emacs-24.3 -o=test.gz
Running on device: Intel(R) FPGA Emulation Device
Compression Ratio 34.2421%
PASSED
```

The next step is to generate static optimization reports for the design. When optimizing, you should inspect these reports to understand the structure of the specialized pipelines being created for your kernels. You can find the reports at `gzip_report.prj/reports/report.html.make reports`.

Finally, you can compile and run the design on FPGA hardware. The optimized compilation will take a few hours.

```
make fpga
```

Performance

Here's how we invoke GZIP:

```
./gzip.fpga <input_file> [-o=<output_file>]
```

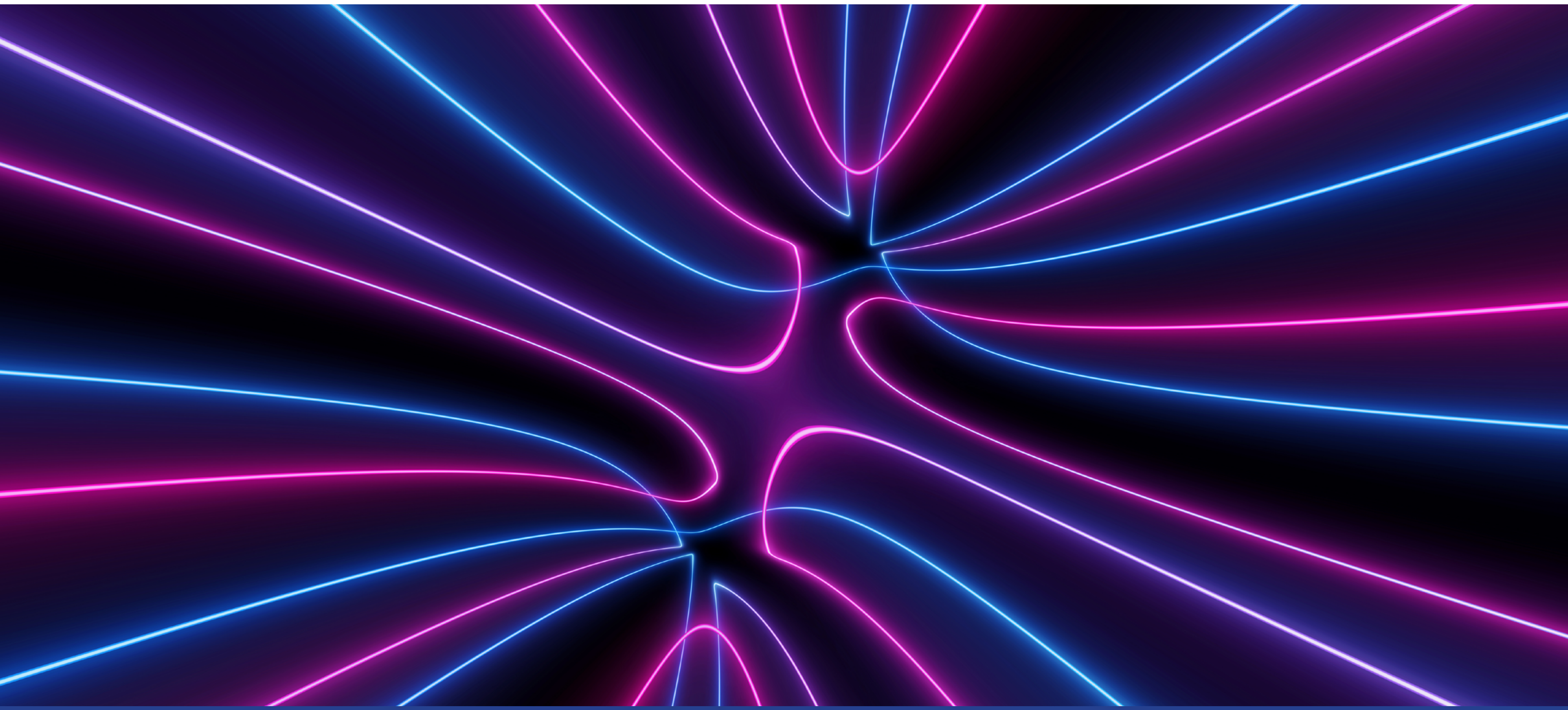
To evaluate performance, the application will call the compression function repeatedly and report on the overall execution time and throughput. Here's some sample output:

```
./gzip.fpga_emu /bin/emacs-24.3 -o=test.gz
Running on device: pac_a10 : Intel PAC Platform (pac_f400000)
3.5 GB/s
Compression Ratio 34.2421%
```

Making FPGAs More Accessible

With oneAPI, FPGAs are more accessible than ever. Spatial architectures open great acceleration opportunities, often in domains that are not embarrassingly parallel. And it's all at your fingertips with the DPC++ compiler and oneAPI.

TEACH YOUR CODE TO BE SMARTER
Download free Intel® Performance Libraries and start creating better, more reliable, and faster applications now.
FREE DOWNLOAD >
intel
Software



IS YOUR GAME GPU-BOUND?

It's Easy to Find Out with a GPU and Device Context Queue Analysis

Oleg Fedyaev, Graphics Software Engineer, Intel Corporation

Computer graphics is an amazing, essential part of our everyday lives—while we work on computers, watch movies, use smartphones, and even drive cars. The performance of graphics processors has dramatically increased over the last 10 years. And the influence of the video gaming industry on this process is hard to underestimate. At the same time, the continuous growth of GPU capabilities is opening new opportunities for game developers, encouraging them to invent breakthrough rendering techniques and effects to gain every possible hardware advantage. But this race between GPU hardware developers and game developers creates a drawback. Often, innovative rendering techniques bump into hardware limits.

In this article, we'll walk through a quick and easy way to see whether your game is CPU-bound using a high-level system overview.

Game Performance Fundamentals

Video game production is expensive. And investing in performance optimization is an important factor contributing to a game project's profitability. Normally, different game genres—action, adventure, strategy, and others—have different performance requirements. If a game seems visually slow, with notable lags or delays in drawing artifacts, it definitely has performance issues that must be addressed.

A formal metric to measure game performance is frame rate, the number of frames rendered per second (FPS). FPS is used for benchmarking and ranking different applications: the higher the FPS, the better. In most cases, this approach is feasible. For example, an action game with a lot of motion doesn't look good if FPS is low.

A modern game is a complex product consisting of multiple components:

- Rendering graphics
- Calculating physics
- Playing sounds
- Executing scripts
- Hosting network
- And more

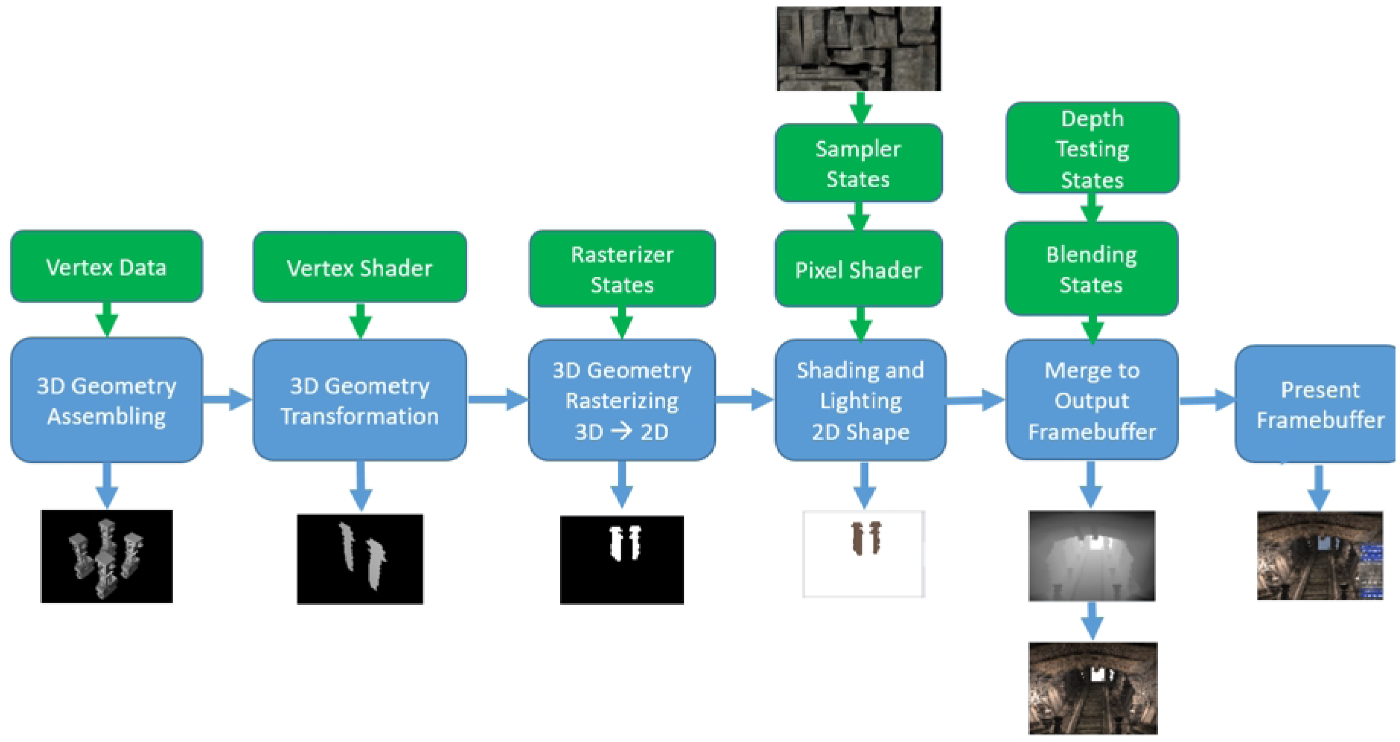
Each component, separately or in combination, can affect game performance. That's why it can be tricky to identify whether an application is GPU- or CPU-bound.

Though it's only one aspect of a game, it's reasonable to start analysis from rendering, since graphics can be crucial to creating the game's unique style, spirit, and atmosphere.

Classic Rendering Pipeline

There's no way to define precisely whether the GPU is a performance bottleneck without understanding the graphics rendering pipeline, the graphics programming model, and the role of a graphics driver in this procedure. A thorough analysis of GPU activity through the whole stack—from the application code to the hardware—requires significant expertise. Fortunately, it's enough to perform a basic performance analysis to see overall GPU utilization without going into great detail.

A rendering pipeline operates with resources and states. Resources bound to the pipeline specify what should be rendered and where. They can consist of geometries, textures, and render targets written in a proper format. Rasterization parameters, testing depth conditions, blending attributes, and other states specify how those resources should be interpreted and processed to generate an image on a screen. Render resources and states are tightly connected with GPU programs, also called shaders, which are executed in different stages of the rendering pipeline (**Figure 1**).



1 Rendering pipeline

The pipeline's classic rendering process takes source data and sequentially modifies it, passing it through the same stages until it reaches the destination. Any rendered object is first transformed in a virtual space, and then projected onto a screen surface. After that, a visible part of that projection is colorized and merged with other rendered objects in a framebuffer.

The graphics programming model is simple. Draw context is configured, opened, and used for submitting rendering commands in the correct order: required resources, states, and programs should be bound to the pipeline before invoking any draw command that orders the pipeline to do a job. This procedure is repeated as many times as needed for objects to be rendered, until a final scene forms in a framebuffer. We can push this to a screen with a buffer swap command. Regardless of whether an application uses OpenGL*, DirectX*, Vulkan* or any other graphics API, this concept stays the same.

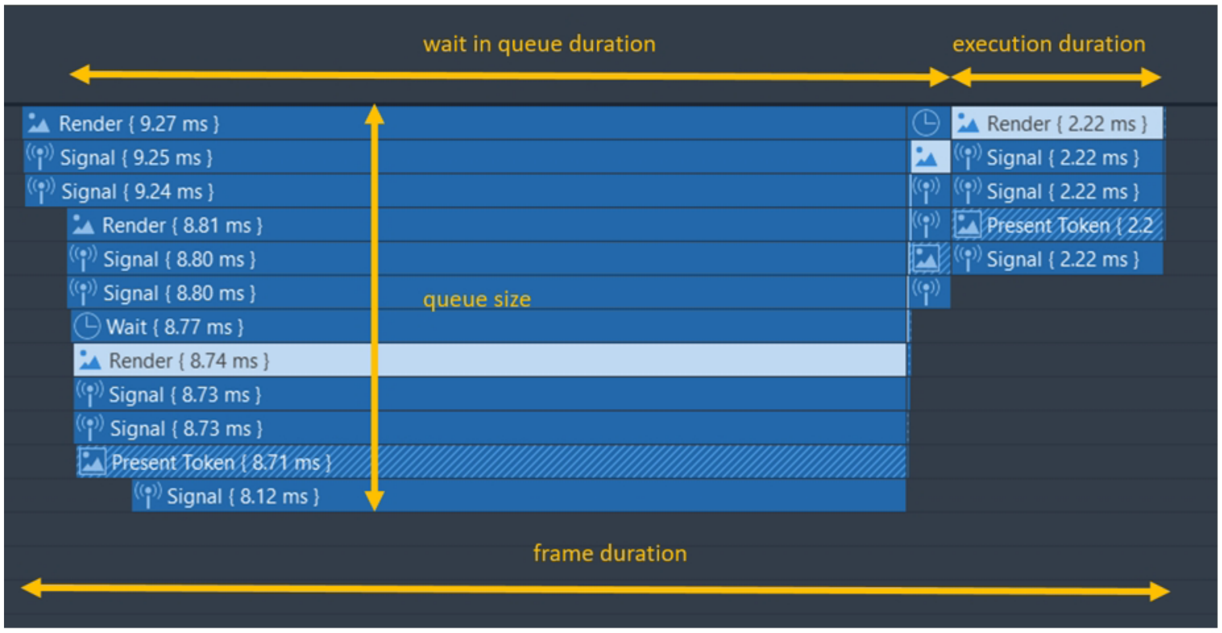
It's now obvious how many complicated operations constitute a single draw. And each operation contributes to the draw duration. Individual draw durations vary, affecting the total frame rendering time. A long frame time may indicate a GPU-bound scenario, which we can confirm or reject after estimating the GPU load based on graphics driver performance markers.

Graphics Driver Activity

A common graphics program works with a graphics driver, but never directly with the GPU. Any time we open a draw context in our application, we implicitly create a corresponding interface to a graphics driver, known as a driver device context. To make rendering possible, the driver must perform a lot of work:

- Release and allocate memory blocks on the GPU
- Upload the resources necessary for rendering from the CPU to the GPU
- Set registers of the GPU execution units
- Upload GPU programs
- Transfer results back to the CPU
- And more

Any time we invoke a sequence of graphics API calls within the application code, the driver translates them into a sequence of commands eligible for the GPU. Commands are not executed on the GPU immediately. Instead, they accumulate in a command buffer. The driver constantly batches a series of commands into packets, and then pushes these packets into a device context queue, scheduling them for execution (**Figure 2**).



2 **Command buffer**

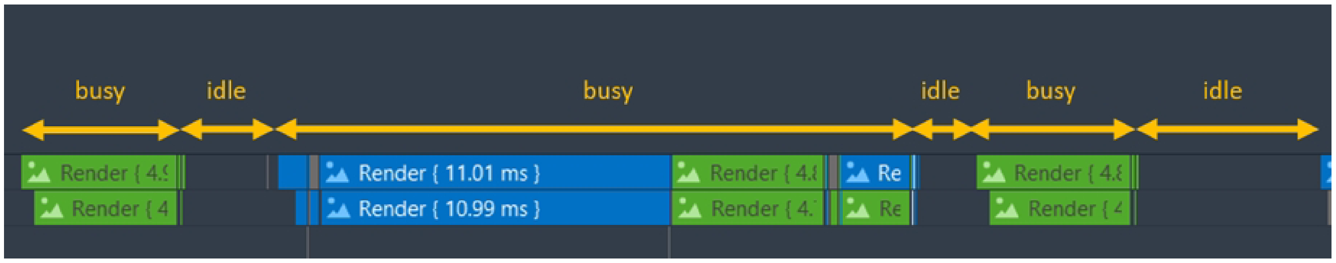
A device context queue can contain different types of packets with different types of commands. The prevailing command type in a packet defines the packet's type. Each packet stays in the queue, waiting until the last command written in the previous packet has been executed on the GPU. (For example, see the selected render packet in **Figure 2**.)

Exploring the device context queue can give us some useful performance insights. For example, a huge queue size usually corresponds to a huge amount of graphics work submitted to the GPU. Long packet execution time may be due to computationally-intensive draw procedures. Long packet waits can be caused by inefficient rendering algorithms or synchronization.

When we've identified all the packets relating to a single frame, we can roughly estimate the frame duration, which we can calculate as a time range by submitting the first command packet in a queue until executing the last command from the last submitted packet within that frame.

However, even if a frame time is long, we can't define whether our application is GPU-bound until we explore a corresponding GPU hardware queue associated with a graphics processor performing rendering. The GPU is a shared resource that can serve multiple applications, rendering graphics simultaneously. Long rendering time may be a result of concurrent execution with another application that acquired the GPU context at the same time.

The hardware GPU queue (**Figure 3**) provides a clear picture of the overall GPU utilization. We can use this queue to identify how busy the GPU is, and which application is rendered at the time.



3 **Hardware GPU queue**

The GPU queue snapshot in **Figure 3** shows at least two simultaneously rendered applications, differentiated by the colors of the command packets. Neither application is GPU-bound. The frame time of the application, highlighted in blue, is not much longer than 11 milliseconds, which corresponds to approximately 80 FPS. And 80 FPS is usually high enough. The green one seems to be a background process with very tiny frames (about 5 milliseconds each). Moreover, the GPU isn't even as busy as it could be, since we can see multiple gaps between executing command packets, corresponding to periods when the GPU is idle.

The concept of analyzing software and hardware queues is quite promising from a reliability perspective. Plus, these queues are easy to build, since we know how to acquire the required performance data.

System Event Tracing

Regardless of whether we work on Windows*, Linux*, macOS*, or any other operating system, we can connect to a system event trace layer, which logs different types of events associated with key execution points within different system modules. Some events are eligible for performance analysis. The graphics driver is no exception. Any time the driver pushes a command packet into a device context queue, uploads a command packet to the GPU, or executes the last command written in a command packet, it submits corresponding events into a system tracing layer so that we can easily acquire them. For example, if we want to build a device context and GPU command packet queues on Windows, we need to capture several events from the Microsoft-Windows-DxgKrnl provider of the Event Tracing for Windows* (ETW*) system.

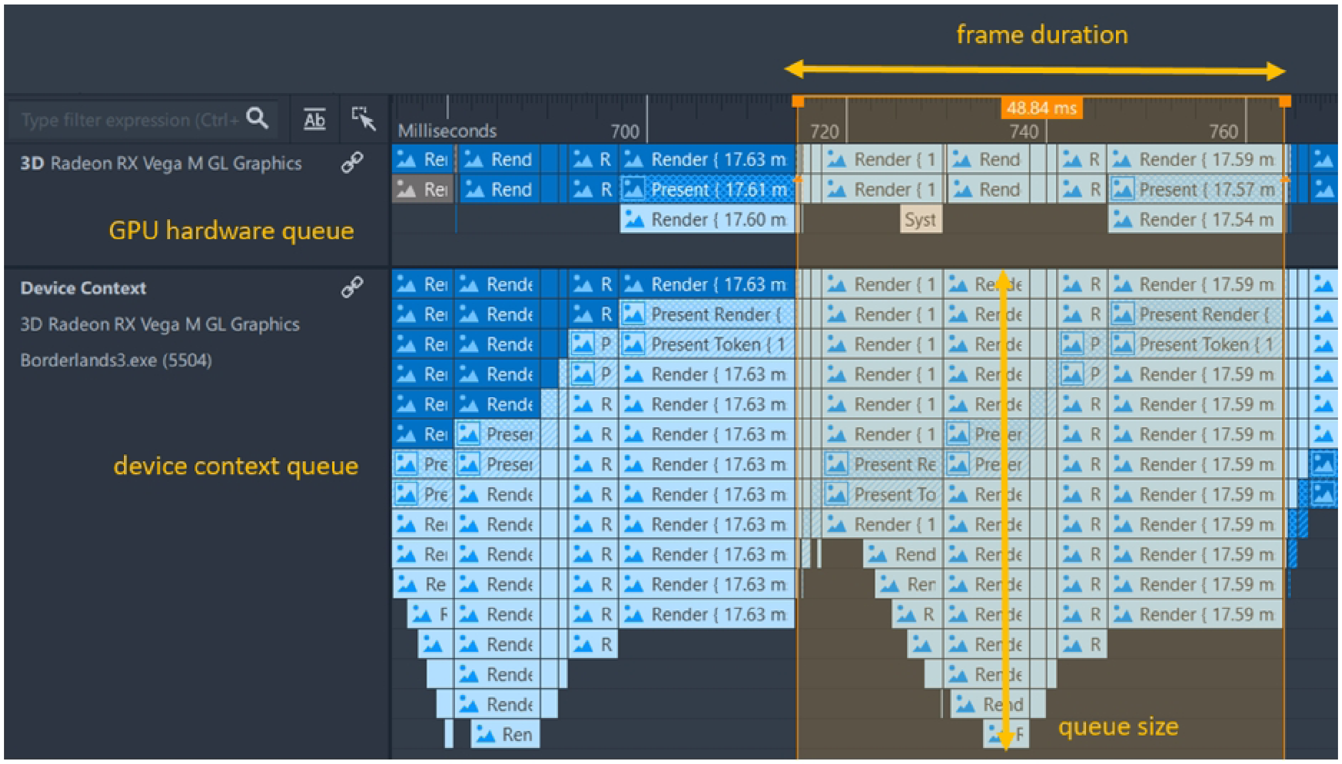
Different attributes encoded into events data enable binding different events together to distinguish the current status of each packet in a queue at any time.

Graphics Application Analysis

System event tracing is well documented and can be used on any platform. There are many tools that can capture or visualize system tracing data. However, the number of tools capable of proper simultaneous analysis of device context and GPU hardware queues is limited. **Intel® GPA Graphics Trace Analyzer** is one of these tools, designed to analyze the performance of graphics applications with different levels of detail, from a high-level system analysis to a single frame per-draw analysis.

Now let's apply what we've learned to a real-life, graphics-intensive game. We can try it on a workstation with a graphics processor in the middle performance range to make our experiment predictable. We'll use the just-released **Borderlands 3***, a well-known game from the first-person shooter genre. We'll run it on the **Intel® NUC Mini PC NUC8i7HVK**, which has two graphics processors: integrated Intel® HD Graphics 630 and discrete AMD Radeon RX Vega M GL*. If we run this game at 2560x1440 resolution adapted for widescreen monitors and switch all graphics options in the game to a high profile, the game engine selects for rendering the most capable graphics processor, which seems to be Radeon Vega on this device.

The first thing that catches our eye after five minutes of play is a lag between changing the state of an input device, such as a mouse or a gamepad, and changing a scene on a screen. Animation of some moving objects also looks a bit ragged. If we capture and open a trace with Intel GPA Graphics Trace Analyzer, we can observe all attributes of the GPU-bound scenario from the first sight at timeline tracks with a device context queue and GPU hardware queue (**Figure 4**).



4 **Device context queue and GPU hardware queue**

The GPU queue has no gaps. It's fully busy, continuously executing commands submitted from the game. The device context queue size is large enough, which means a lot of graphics work is prepared and waiting for rendering. We can also measure frame duration, selecting all command packets executed on the GPU within a single frame. The frame duration of about 48.8 milliseconds corresponds to approximately 21 FPS, which is definitely insufficient for this action game. Games like this one usually require 60+ FPS to achieve maximum game experience.

Analyzing GPU-bound scenarios using a high-level system overview, and exploring software and hardware queue breakdowns, gives us several benefits. This analysis is quick and accurate. It doesn't require graphics expertise or depend on the type of graphics API used for rendering. It also works on any platform where we can capture corresponding performance events from system tracing.

Learn More

- **Intel® Graphics Performance Analyzers**
- **Intel® Graphics Trace Analyzers**
- **Intel® Graphics Performance Analyzers Cookbook**

DECODE YOUR TECH FUTURE

intel Software

Welcome to Tech.Decoded, the Knowledge Hub for Developers

You'll find an always-growing library of information curated to help you get the most out of modern hardware. Boost your competitive edge. And get to market faster.

Get Expert Insights.
Watch tech forecasters and visionaries explore today's tech landscape: code modernization, systems and IoT, data science, and more.

Dig Deeper.
Learn how to get every last ounce of performance from your code with on-demand webinars covering today's most important strategies, practices, and tools.

Put it All to Work in your Code.
Use short videos and articles to understand the how-to's of key programming tasks using specific development tools.

EXPLORE TECH.DECODED NOW >

NEW THREADING CAPABILITIES IN JULIA* V1.3

Unleashing the Full Power of Modern CPUs

Jameson Nash and Jeff Bezanson, Julia Computing, Inc., and Kiran Pamnany, Caltech

Taking advantage of multicore processors is a crucial capability for any modern programming language. Programmers need the best possible throughput for their applications, so we're going to show how the new multithreading runtime in Julia* v1.3 unleashes the full power of a modern CPU with minimal hassle.

One of our key considerations is reducing the programmer's burden. Julia provides a range of modern primitives designed to compose effectively. We'll discuss some of the tradeoffs we make to try to simplify the mental model for the programmer. One important design principle of Julia is to make common tasks easy and difficult tasks possible. This is demonstrated in multiple aspects of the language, e.g.:

- Automatic memory management
- Combining functions, objects, and templates into a single dispatch mechanism
- Optional type inference for performance

We now extend this to parallelism. By building on the language's existing concurrency mechanism, we've added parallel capabilities that preserve the (relative) simplicity of single-threaded execution for existing code, while allowing new code to benefit from multithreaded execution. This work has been inspired by parallel programming systems such as **Threading Building Blocks**.

In this paradigm, any piece of a program can be marked for parallel execution, and a task will be started to automatically run that piece of code on an available thread. A dynamic scheduler decides when and where to launch tasks. This model of parallelism has many helpful properties. We see it as somewhat analogous to garbage collection. With garbage collection, you freely allocate objects without worrying about when and how they're freed. With task parallelism, you freely spawn tasks—potentially millions of them—without worrying about when and where they eventually run.

The model is portable and free from low-level details. The programmer doesn't need to manage threads, or even know how many processors or threads are available. The model is nestable and composable. Parallel tasks can be started that call library functions that themselves start parallel tasks—and everything works correctly. This property is crucial for a high-level language where a lot of work is done by library functions. The programmer can write serial or parallel code without worrying about how the underlying libraries are implemented. This model isn't limited to Julia libraries, either. We've shown that it can be extended to native libraries such as FFTW*, and we are working on extending it to OpenBLAS*.

Running Julia with Threads

Let's look at some examples using Julia v1.3 launched with multiple threads. To follow along on your own machine, you'll need to download the latest Julia release (currently v1.3.0) from **https://julialang.org/downloads**. Run `./julia` with the environment variable `JULIA_NUM_THREADS` set to the number of threads to use. Alternatively, after installing Julia, follow the steps at **http://docs.junolab.org/latest/man/installation/** to install the Juno IDE*. It will automatically set the number of threads based on available processor cores. It also provides a graphical interface for changing the number of threads.

We can verify that threading is working by querying the number of threads and the ID of the current thread:

```
julia> Threads.nthreads()
4

julia> Threads.threadid()
1
```

Tasks and Threads

A visual way to demonstrate that threads are working is to watch the scheduler picking up work in semi-random, interleaving orders. Previous versions of Julia already had a '`@threads for`' macro which would split a range and run a portion on each thread with a static schedule. So in the range below, thread 1 would run items 1 and 2; thread 2 would run items 3 and 4; and so on.

```
bash$> JULIA_NUM_THREADS=8 julia <<EOF
Threads.@threads for i = 1:12
    println(i, " on thread ", Threads.threadid())
end
EOF
1 on thread 1
3 on thread 2
12 on thread 8
9 on thread 5
7 on thread 4
2 on thread 1
4 on thread 2
5 on thread 3
8 on thread 4
11 on thread 7
10 on thread 6
6 on thread 3
```

It's now possible to run the same program with a completely dynamic schedule. We use the new `'@spawn'` macro with the existing `'@sync'` macro to delineate the work items.

```
bash$> JULIA_NUM_THREADS=8 julia <<EOF
@sync for i = 1:12
    Threads.@spawn println(i, " on thread ", Threads.threadid())
end
EOF
2 on thread 5
3 on thread 4
8 on thread 7
6 on thread 5
12 on thread 7
7 on thread 6
9 on thread 8
10 on thread 5
4 on thread 3
1 on thread 2
5 on thread 1
11 on thread 1
```

Now, let's look at some more practical examples.

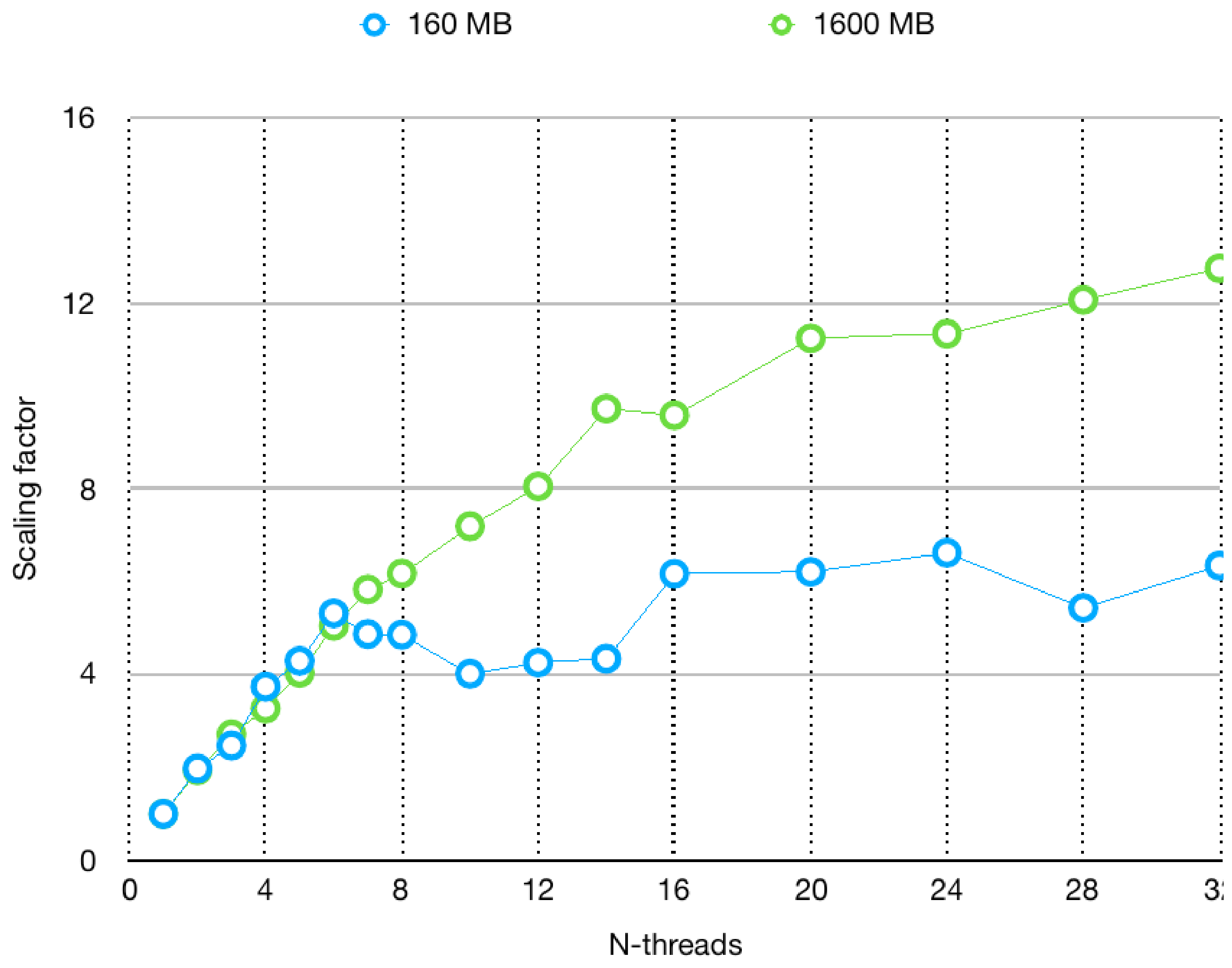
Parallel Merge Sort

The classic merge sort algorithm shows a nice performance benefit from using multiple threads. This function will create *O(n)* subtasks, which will sort independent portions of the array before merging them into a final sorted copy of the input. We use here the ability of each task to return a value via `fetch`.

```
# perform a merge sort on `v` using parallel threads
function psort(v::AbstractVector)
    hi = length(v)
    if hi < 100_000 # below some cutoff, run in serial
        return sort(v, alg = MergeSort)
    end

    # split the range and sort the halves in parallel recursively
    mid = (1 + hi) >>> 1
    half = Threads.@spawn psort(view(v, 1:mid))
    right = psort(view(v, (mid + 1):hi))
    left = fetch(half)::typeof(right)

    # perform the merge on the result
    out = similar(v)
    pmerge!(out, left, right)
    return out
end

function merge!(out, left, right)
    ll, lr = length(left), length(right)
    @assert ll + lr == length(out)
    i, il, ir = 1, 1, 1
    @inbounds while il <= ll && ir <= lr
        l, r = left[il], right[ir]
        if isless(r, l)
            out[i] = r
            ir += 1
        else
            out[i] = l
            il += 1
        end
        i += 1
    end
    @inbounds while il <= ll
        out[i] = left[il]
        il += 1
        i += 1
    end
```

```
 @inbounds while ir <= lr
        out[i] = right[ir]
        ir += 1
        i += 1
    end
    return out
end

function pmerge!(out, left, right)
    ll, lr = length(left), length(right)
    @assert ll + lr == length(out)
    if length(out) < 100_000
        # below some threshold, just do the merge
        merge!(out, left, right)
    else
        # split the larger chunk in half, then binary search the smaller half to split it
        if ll > lr
            jl = ll ÷ 2
            # stable sort: find the last entry in right strictly smaller than l
            jr = searchsortedfirst(right, left[jl]) - 1
        else
            jr = lr ÷ 2
            # stable sort: find the last entry in left not bigger than r
            jl = searchsortedlast(left, right[jr])
        end
        @sync begin
            let left = view(left, 1:jl),
                right = view(right, 1:jr),
                out = view(out, 1:(jl + jr))
                Threads.@spawn pmerge!(out, left, right)
            end
            let left = view(left, (jl + 1):ll),
                right = view(right, (jr + 1):lr),
                out = view(out, (jl + jr + 1):length(out))
                pmerge!(out, left, right)
           end
        end
    end
    nothing
end
```

Figure 1 shows how adding more threads affects scaling. Since we're using in-process threads, we could further optimize by mutating the input in place and reusing the work buffers for additional performance.



1 **Scaling ratios of `psort` on a server with 40 hyperthreads (two Intel® Xeon® Silver 4114 processors @ 2.20GHz)**

While not demonstrated here, `fetch` would also automatically propagate exceptions from the child task.

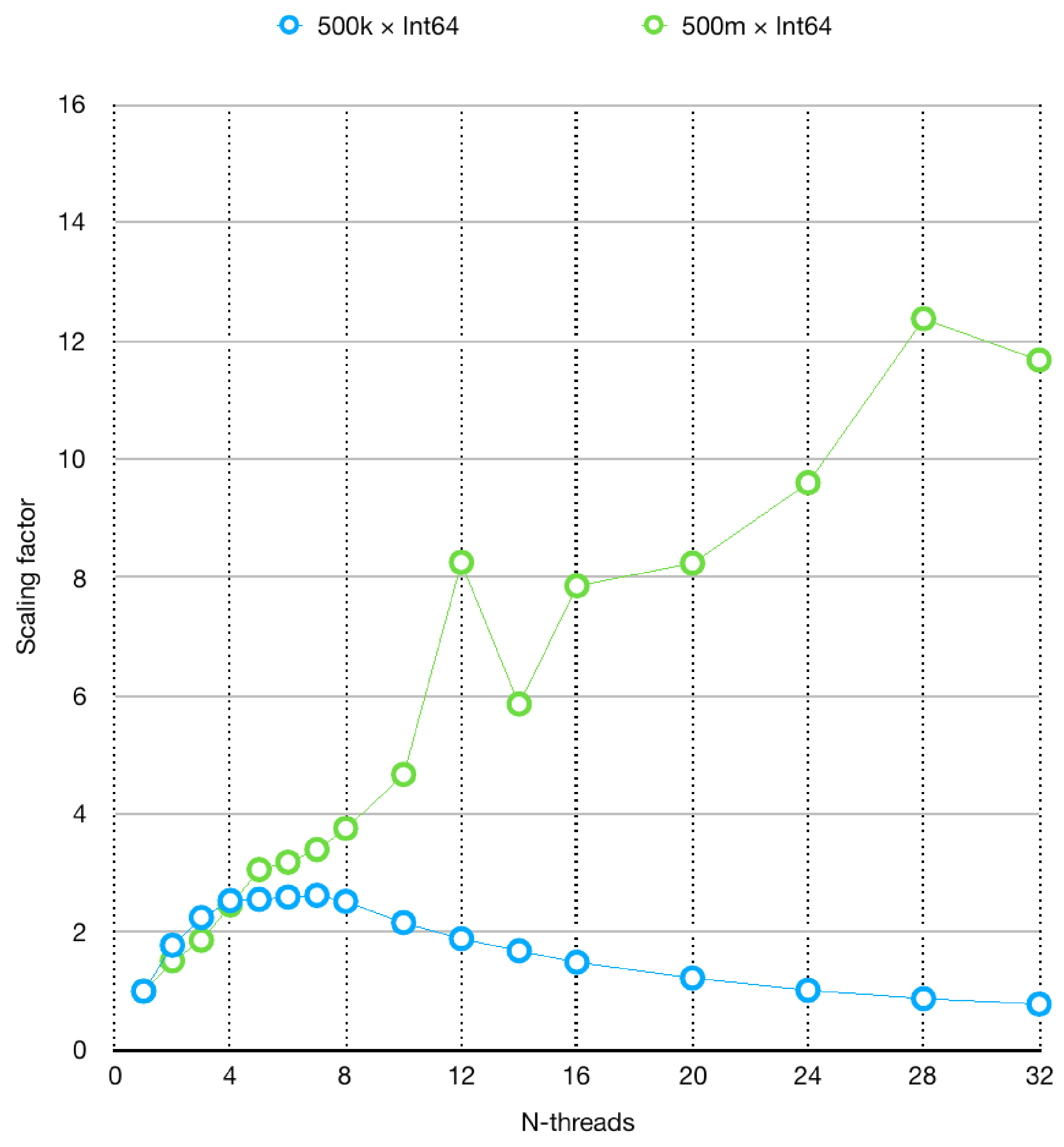
Parallel Prefix

Prefix sum (also known as "scan sum") is another classic problem that can benefit nicely from multiple threads. The parallel version of the algorithm computes partial sums by arranging the work into two trees. The short implementation below can take advantage of all cores and SIMD units available on the machine:

```
using .Threads: @threads
function prefix_threads!(⊕, y::AbstractVector)
    l = length(y)
    k = ceil(Int, log2(l))
    # do reduce phase
    for j = 1:k
        @threads for i = 2^j:2^j:min(l, 2^k)
            @inbounds y[i] = y[i - 2^(j - 1)] ⊕ y[i]
        end
    end
    # do expand phase
    for j = (k - 1):-1:1
        @threads for i = 3*2^(j - 1):2^j:min(l, 2^k)
            @inbounds y[i] = y[i - 2^(j - 1)] ⊕ y[i]
        end
    end
    return y
end

A = fill(1, 500_000)
prefix_threads!(+, A)
```

Figure 2 shows how adding more threads affects scaling.

Several features of Julia combine to make it particularly easy to express this simple yet performant implementation. Under the hood, the system automatically compiles versions of the function optimized for different types of arguments. The compiler can also automatically specialize the function for a specific CPU model, both ahead-of-time and just-in-time. Julia ships with a "system image" of code precompiled for a reasonable range of CPUs, but if the processor used at runtime supports a larger feature set, the compiler will automatically generate better-tailored code. Meanwhile, the threading system adapts to the available cores by dynamically scheduling work.



2 **Scaling ratios of `prefix_threads` on a server with 40 hyperthreads (two Intel® Xeon® Silver 4114 processors @ 2.20GHz)**

Parallel-Aware APIs

Several operations in application code must be made thread-aware to be used safely in parallel. These user-facing APIs include:

- **Concurrency basics:** `Task`, and associated functions including `schedule, yield,` and `wait`
- **Mutexes:** `ReentrantLock` and `Condition` variables, including `lock, unlock,` and `wait`
- **Synchronization primitives:** `Channel, Event, AsyncEvent,` and `Semaphore`
- **I/O and other blocking operations:** Including `read, write, open, close,` and `sleep`
- **Experimental Threads module:** Various building blocks and atomic operations

Scheduler Design

A prototype implementation of the `partr` scheduler was first written for us in C by Kiran Pamnany while at Intel in late 2016[1], following research on cache-efficient scheduling[2]. The goal of this work was composition of threaded libraries with a globally depth-first work ordering. `partr` implements this using an approximate priority queue, where the priorities are set equal to the thread ID of the thread launching a task.

Foreign Libraries

An important motivation for this work was our desire to better support multithreaded libraries without CPU oversubscription killing performance due to cache-thrashing and frequent context switching. Previously, the only options were for the user to decide up-front to limit Julia to N threads, and to tell the threaded library (such as libfftw or libblas) to use M ÷ N cores. The most common choices are probably 1 and M, so only part of the application can benefit from multiple cores. However, given our ability to quickly create and run work items in our thread pool, we're looking at how to let external libraries integrate with our thread pool. This is an ongoing area of exploration as we get feedback on the performance and API needs of various libraries.

We've successfully adapted FFTW to run on top of our threading runtime instead of its own (a Pthreads-based workpool). This took us only a few hours. (We were fortunate to be able to enlist the help of that library's author.) Without any performance tuning (yet), it got competitive performance results. We learned important lessons to tightly optimize our scheduler latency, which is now ongoing work to achieve exact performance parity. Even with some overhead imposed by generality, however, we expect that the ability to compose thread-aware users and achieve resource sharing from the partr scheduler will make this an overall improvement in program operation.

Looking to the Future

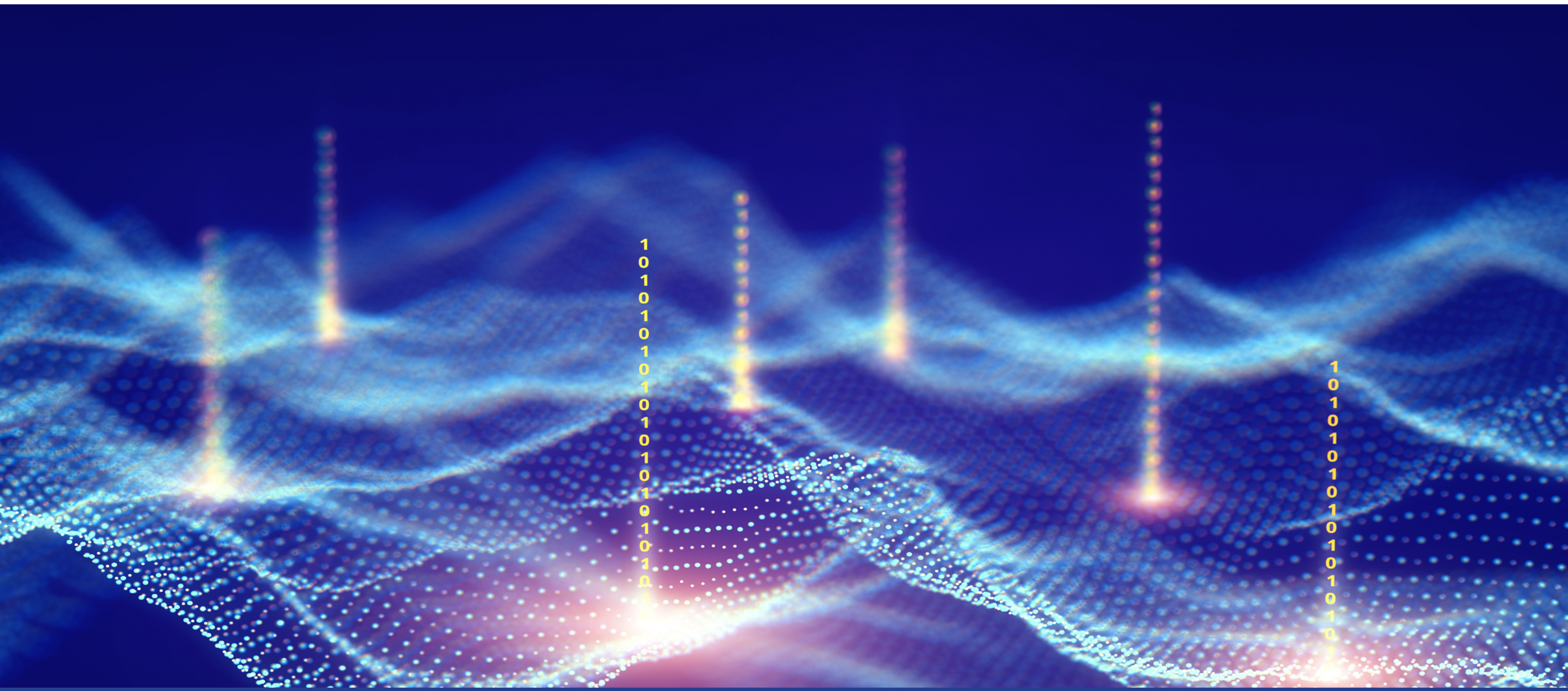
Julia's approach to multithreading combines many previously known ideas in a novel framework. While each in isolation is useful, we believe that—as is so often the case—the sum is greater than the parts. From this point, we hope to see a rich set of composable parallel libraries develop within the Julia ecosystem.

References

1. **https://github.com/kpamnany/partr**
2. Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. Scheduling Threads for Constructive Cache Sharing on cmps. In Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '07, pages 105–115, New York, NY, USA, 2007. ACM.

Deliver uncompromised performance for diverse workloads across multiple architectures with oneAPI.

LEARN HOW >



FAST GRADIENT BOOSTING TREE INFERENCE FOR INTEL® XEON® PROCESSORS

How to Boost Prediction Quality and Performance Using GBT in Intel® Data Analytics Acceleration Library

Kirill Shvets, Machine Learning Engineer, and Egor Smirnov, Software Engineering Manager, Intel Corporation

Gradient Boosted Trees* (GBT*)[1] is an accurate and efficient machine learning (ML) algorithm for classification and regression tasks. There are many GBT implementations, but perhaps the most popular is the XGBoost*[2] library. Our previous article, "Accelerating XGBoost for Intel® Xeon® Processors" (*The Parallel Universe*, **issue 38**),[3] reported a significant improvement in CPU-based training for XGBoost. The performance for the **Intel® Data Analytics Acceleration Library (Intel® DAAL)**[4] was shown to be even better. ML inference is just as important as training because end users need model predictions as quickly as possible. That's the focus of this article.

Performance Comparison

In this article, we'll compare inference performance for the following GBT implementations:

- **Forest Inference Library (FIL)**[5], an open-source library in RAPIDS cuML*[6] that provides GPU-accelerated inference for boosted decision tree models
- **XGBoost** CPU/GPU inference (master branch)
- **Intel DAAL** (version 2019, Update 5), a highly-optimized open-source library for Intel® platforms

Real-world datasets were used for performance analysis: Bosch*, Epsilon*, and Mortgage* (Q1'2000 subset). We shuffled each dataset and randomly selected the following subsets for the prediction stage:

- **Bosch**[7] (968 features, 100K total number of observations, ~1.1M for prediction)
- **Epsilon**[8] (2,000 features, 400K total number of observations, 100K for prediction)
- **Mortgage**[9] Q1'2000 (45 features, ~9M total observations, 1M for prediction)

We chose model sizes sufficient to achieve best accuracy on the selected datasets (Appendix A). For performance measurements, we used AWS EC2* instances:

- **CPU:** c5.metal (2nd generation **Intel® Xeon® Scalable processors**, 2 sockets, 24 cores per socket)
- **GPU:** p3.2xlarge (NVIDIA* Tesla* V100)

For details, see the Configuration section below.

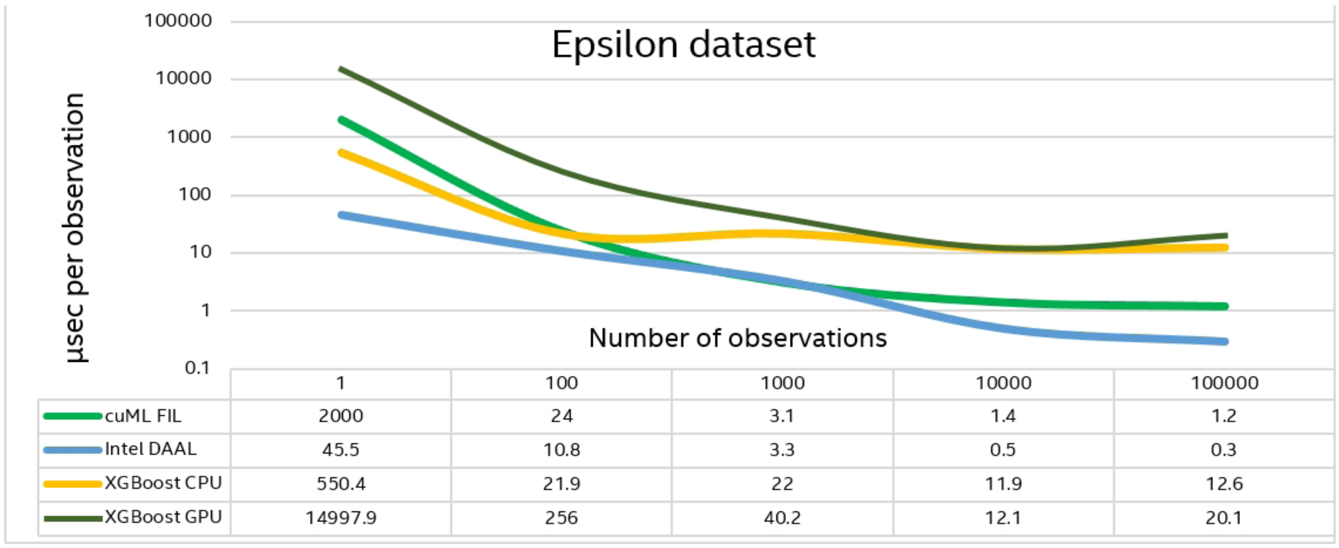
It's common in ML to apply the inference model to one data observation at a time (i.e., the batch size is set to 1). GBT inference can also be applied this way, but to be comprehensive, we'll consider varying batch sizes.

Figure 1, **Figure 2**, and **Figure** 3 show:

- **The CPU implementation of XGBoost** (yellow line) is competitive with the GPU implementation (dark green line), and even outperforms it on smaller batches.
- **FIL (green line) is faster than stock XGBoost CPU/GPU** in most cases (medium and large batch sizes), while Intel DAAL performs even better.
- **The Intel DAAL implementation (blue line) significantly outperforms FIL** (green line) on a one-row case.
- **The Intel DAAL implementation (blue line) has better performance** for medium and larger batch sizes.

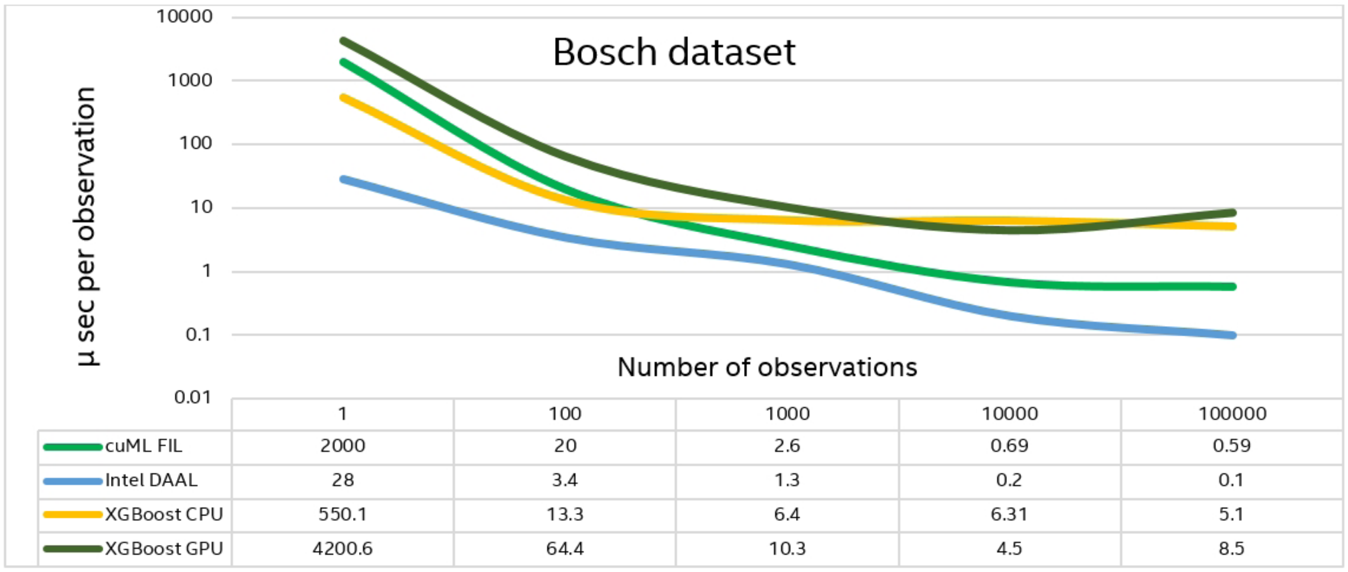
For XGBoost GPU and CPU performance measurements, input data was provided in the native format, XGBoost DMatrix*. For prediction with GBT in Intel DAAL, we passed NumPy* contiguous arrays.

For internal implementation reasons, the cuDF* data format isn't performance-oriented for FIL. This meant that we had to provide NumPy contiguous arrays as an input, since it's a more efficient—and faster—way to load data on a device than an external load (e.g., the CuPy* library that we also used). However, in real end-to-end ML workloads on the GPU, it's more common to do ETL with cuDF and input a cuDF DataFrame without data conversion (i.e., without any additional overhead). **Figure 4** shows that the cuDF format for FIL inference is dramatically slower than FIL with the NumPy data format, especially compared to Intel DAAL. We get expected performance in end-to-end machine learning applications.



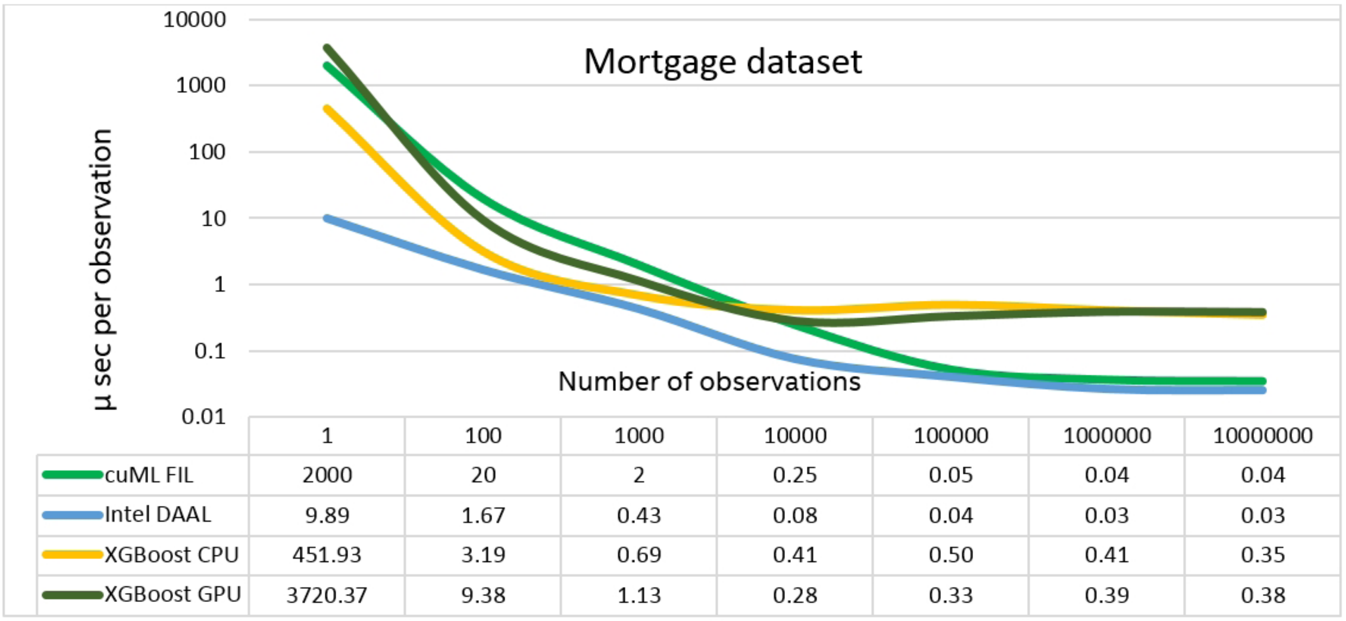
	1	100	1000	10000	100000
cuML FIL	2000	24	3.1	1.4	1.2
Intel DAAL	45.5	10.8	3.3	0.5	0.3
XGBoost CPU	550.4	21.9	22	11.9	12.6
XGBoost GPU	14997.9	256	40.2	12.1	20.1

1 **GBT inference performance for the Epsilon dataset (lower values of µsec are better)**



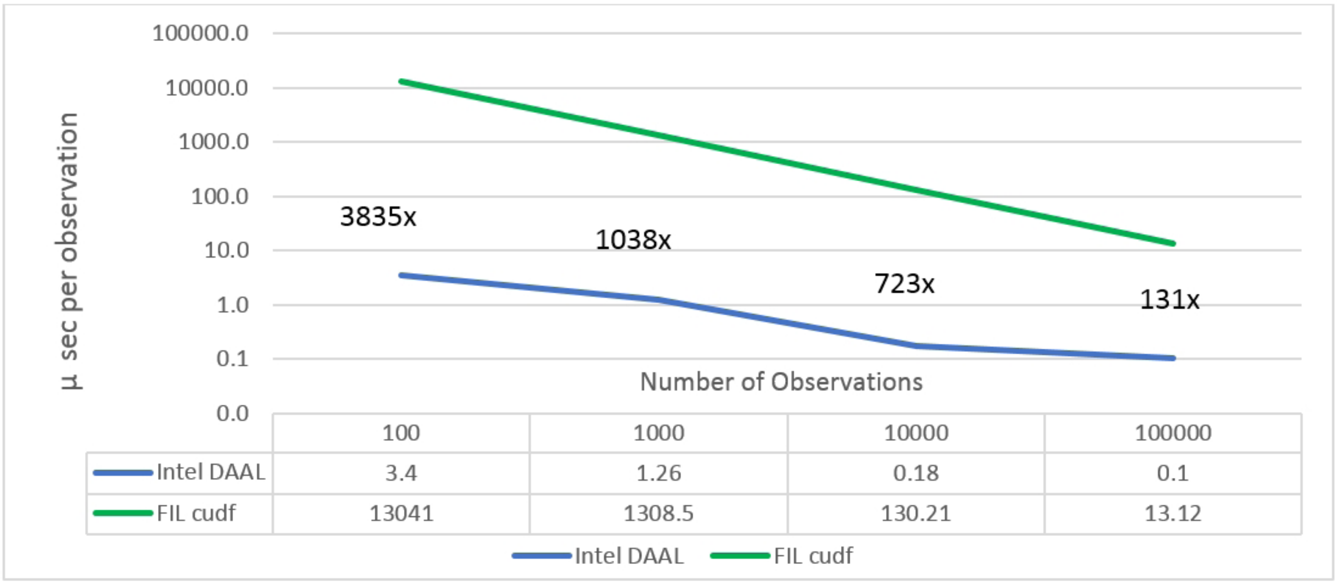
	1	100	1000	10000	100000
cuML FIL	2000	20	2.6	0.69	0.59
Intel DAAL	28	3.4	1.3	0.2	0.1
XGBoost CPU	550.1	13.3	6.4	6.31	5.1
XGBoost GPU	4200.6	64.4	10.3	4.5	8.5

2 **GBT inference performance for the Bosch dataset (lower values of µsec are better)**



	1	100	1000	10000	100000	1000000	10000000
cuML FIL	2000	20	2	0.25	0.05	0.04	0.04
Intel DAAL	9.89	1.67	0.43	0.08	0.04	0.03	0.03
XGBoost CPU	451.93	3.19	0.69	0.41	0.50	0.41	0.35
XGBoost GPU	3720.37	9.38	1.13	0.28	0.33	0.39	0.38

3 **GBT inference performance for the Mortgage dataset (lower values of µsec are better)**



	100	1000	10000	100000
Intel DAAL	3.4	1.26	0.18	0.1
FIL cudf	13041	1308.5	130.21	13.12

4 **GBT inference on FIL cuDF, Intel DAAL throughput, Bosch (lower values of µsec are better)**

We can draw the following conclusions from our performance measurements:

- **Inference on Intel DAAL is significantly faster** than other implementations when the batch size is equal to 1, which is one of the most important use-cases.
- **Inference on Intel DAAL is faster even for bigger batches** and for reasonably-sized models.
- **XGBoost CPU and GPU inference** is approximately the same for all datasets.
- **The cuDF format** (commonly used in end-to-end applications) for FIL inference is dramatically slower than Intel DAAL inference.

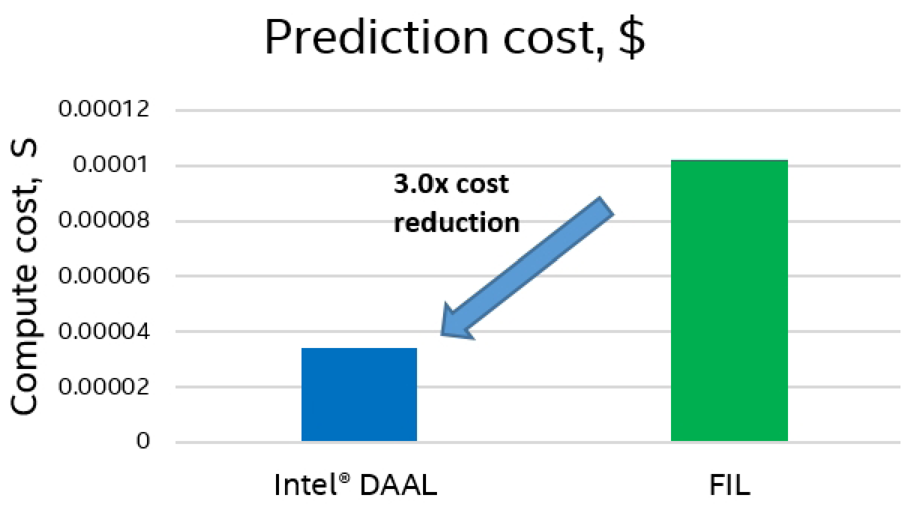
Cost Analysis

We selected the most efficient CPU and GPU implementations to determine prediction cost: Intel DAAL and cuML FIL. The following AWS EC2 instances were used in our experiments (cost as of December 2019):[10]

- **CPU:** c5.metal, $4.08 per hour
- **GPU:** p3.2xlarge, $3.06 per hour

Table 1 shows an example of prediction cost computation for the Epsilon* dataset (100K observations). **Figure 5** shows a prediction cost comparison for Intel DAAL versus FIL.

Library	Time, ms	Instance cost, $/h	Inference cost (Time x Instance cost), $
cuML FIL	120	3.06	$10.2 * 10^{-5}$
Intel DAAL	30	4.08	$3.4 * 10^{-5}$



5 Prediction cost comparison for Intel DAAL versus FIL

The same cost calculations for other datasets show:

- **Bosch dataset:** 4.4x improvement with Intel DAAL versus cuML for prediction cost
- **Mortgage data set:** Intel DAAL is on par with cuML

Compared to the GPU implementations, GBT inference with Intel DAAL on the CPU is faster and less expensive.

Functionality Analysis

We compared available functionality in XGBoost, Intel DAAL, and FIL and summarized it in **Table 2**.

Table 2. **Functionality in XGBoost, Intel DAAL, and FIL**

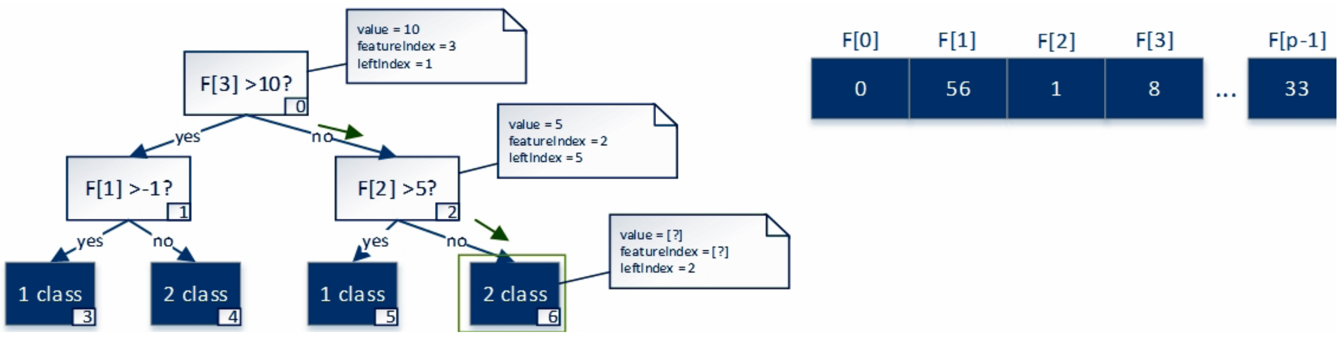
Support	XGBoost	Intel DAAL	FIL
Binary classification	Yes	Yes	Yes
Multiclass classification	Yes	Yes	No
Regression	Yes	Yes	Yes
Float64 input format	Yes	Yes	No
Float32 input format	Yes	Yes	Yes

Due to limitations in FIL, all performance comparisons were provided in single-precision floating-point format and using binary classification and regressions datasets.

What Makes Intel DAAL Faster?

To maximize the utilization of modern **Intel® Xeon® processors**, Intel DAAL applies the **Intel® Advanced Vector Extensions 512 (Intel® AVX-512)** vector instruction set for fast GBT inference (**Figure 6**). The main operations in GBT inference are:

- **Data comparison** for evaluating conditions in each split-node
- **Random memory accesses** (read) to tree nodes and single features from predicted observations



6 **GBT inference**

To process an observation via one tree, we used this idea until reaching a leaf-node: on the current node of the tree, we compared a certain feature value from the observation ‘*F*[*node.featureIndex*]’ with the split value '*node.value*' from the node. The index of the next node is defined as:

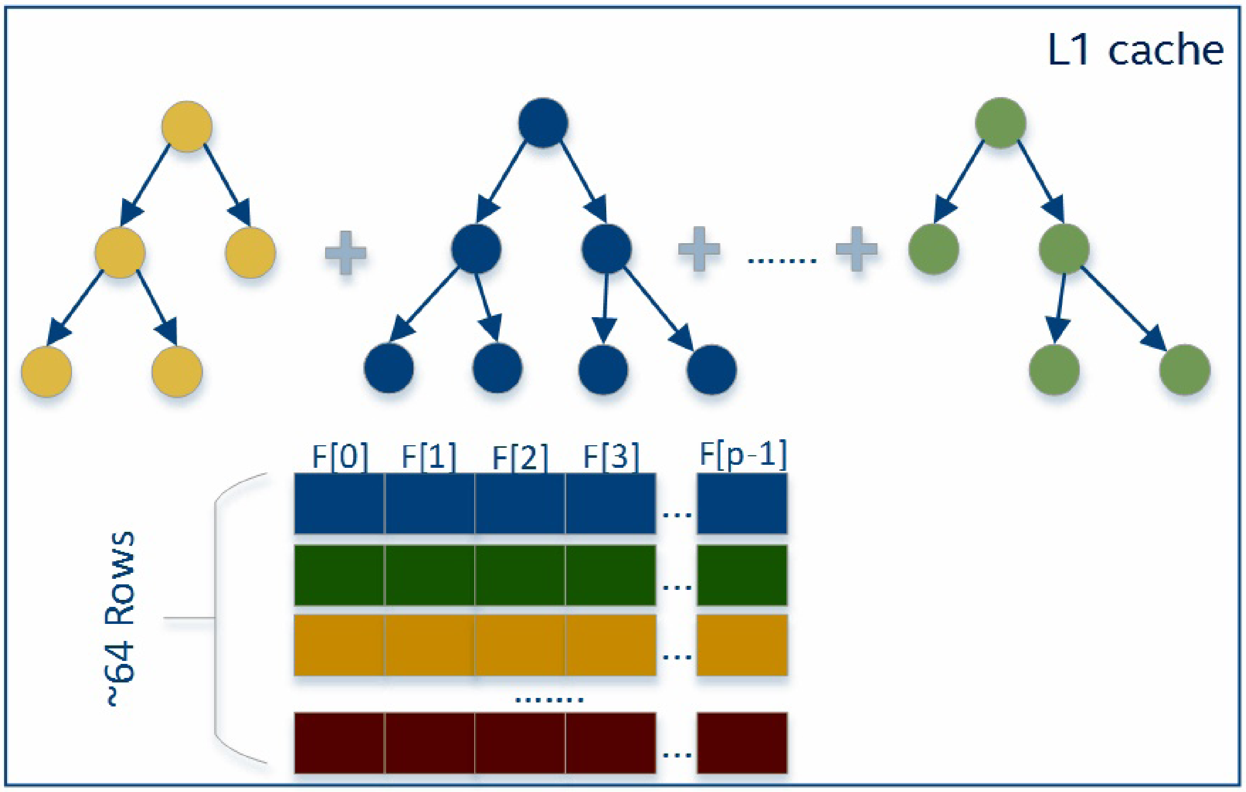
$$node.leftIndex + \ intercept, \text{where } intercept = \begin{cases} 0, if \ F[node.featureIndex] > node.value \\ 1, if \ F[node.featureIndex] \le node.value \end{cases}.$$

The main inference operations, comparison, and random memory accesses can be efficiently implemented with the `vpgatherd{d,q}` and `vcmpp{s,d}` Intel AVX-512 vector instructions. To apply these instructions, search the tree for a few observations (a block of rows) at the same time for each tree:

- **Gather information**, splitting values ‘*value*’ and feature indices ‘*featureIndex*’ from certain tree nodes for each observation in the block of rows.
- **Apply vector comparisons** between ‘*value*’ and *F*[*featureIndex*] for the observations.
- **Compute indices** for the next nodes for each observation based on the comparison.

We do this for each observation until we achieve the leaf nodes in the tree.

The performance of the algorithm depends on the efficiency of memory accesses and memory bandwidth. Intel DAAL uses smart data blocking for tree structures to improve temporal cache locality, a subset of trees, and a block of observations that fits in the L1-data cache size. After this, we implement the required prediction for the observations with the subset of trees. As a result, most data accesses lead to L1 data cache hits with the highest memory bandwidth (**Figure 7**).



7 **Memory bandwidth**

Faster Performance

Summarizing the results of our performance comparisons, we can conclude that we can achieve the same prediction quality much faster performance using GBT in Intel DAAL on a CPU compared to other implementations. The inference with Intel DAAL is significantly faster for online inferencing (i.e., when the batch size equals 1). This is an important use case in applied ML, since it's important to minimize the time required for handling the user-supplied data and predicting the outcome (e.g., object detection in autonomous driving), and there's no possibility of accumulating a bigger batch of data.

For general cases when data was accumulated in bigger batches, Intel DAAL on the CPU also outperforms both the FIL GPU and XGBoost. For maximum available batch size, we have a relative speedup for Intel DAAL versus FIL: 4x for Epsilon, 5.9x for Bosch, and 1.4x for Mortgage.

Besides fast performance, Intel DAAL provides a user-friendly Python* interface, with code for inference that's simple and short.

In computational cost calculations, we saw that for the same payload, the cost for CPU-based inferencing with Intel DAAL is, on average, 2.85x times cheaper than for FIL.[10]

Code Examples

For XGBoost:

```
import xgboost as xgb

# x_train, labels, x_test = ...

# Train model with XGBoost
model_xgb = xgb.train(..., xgb.DMatrix(x_train))

# Generate predictions (as a NumPy array)
predicts = model_xgb.predict(xgb.DMatrix(x_test))
```

For Intel DAAL:

```
import daal4py as d4p

# x_train, labels, x_test, n_classes = ...

# Train model with Intel® DAAL
train_algo = d4p.gbt_classification_training(n_classes, ...)
train_result = train_algo.compute(x_train, labels)

# Generate predictions (as a NumPy array)
pred_alg = d4p.gbt_classification_prediction(n_classes)
predicts = pred_alg.compute(x_test, train_result.model)
```

For FIL:

```
from cuml import ForestInference

# Load the classifier previously saved with xgboost model_save()
model_path = "xgb.model"
fm = ForestInference.load(model_path, output_class=True)

# Generate predictions (as a gpu array)
fil_preds_gpu = fm.predict(X_test.astype('float32'))
```

References

1. Tianqi Chen and Carlos Guestrin. **XGBoost: A Scalable Tree Boosting System**. In 22nd SIGKDD Conference on Knowledge Discovery and Data Mining, 2016.
2. **XGBoost Library**
3. **The Parallel Universe, Issue 38**
4. **Intel DAAL**
5. **Forest Inference Library**
6. **cuML**
7. **Bosch dataset**
8. **Epsilon dataset**
9. **Mortgage dataset**
10. **AWS EC2 Pricing as of 12/02/2019**

Configuration

Algorithm Parameters

While the number of trees and maximum tree depth parameters varied for each dataset, all other parameters were fixed (**Table 3**). Performance data were collected in November 2019.

Table 3. Parameters for XGBoost and Intel DAAL tests

XGBoost	Intel DAAL
Fptype: float32	Fptype: float32
'max_bin': 256	'maxBins': 256
‘learning_rate': 0.1	'shrinkage': 0.1
'reg_lambda': 1	'lambda_': 1

For FIL inference, we used a pre-trained XGBoost model.

Due to FIL limitations, performance was measured only for single-precision floating-point format (float32), since FIL doesn’t support double format currently.

Hardware Configuration

- **CPU configuration:** c5.metal AWS EC2 instance (2nd generation Intel® Xeon® Scalable processors): 2 sockets, 24 cores per socket, HT:on, Turbo:on OS: Ubuntu 18.04.2 LTS, total memory of 193 GB (12 slots/16GB/2933 MHz).
- **GPU configuration:** p3.2xlarge AWS EC2 instance: Intel® Xeon® E5-2686 v4 processor @ 2.30GHz, 1 socket, 4 cores, HT:on, Turbo:on; GPU: Tesla* V100-SXM2-16G (driver version: 410.104, CUDA* version: 10.0), OS: Ubuntu* 18.04.2 LTS, total memory: 61 GB (4 / 13312 MB).

Software Configuration

- RAPIDS FIL (Forest Inference Library) – version 0.9
- Intel® DAAL - version 2019 Update 5
- XGBoost – master (ef9af33a000f09dbc5c6b09aee133e38a6d2e1ff)

Other software: Python 3.6, Numpy 1.16.4, Pandas 0.25, Scikit-lean 0.21.2.

Appendix

Accuracy on a tested subset generally correlates with two major model parameters: the number and depth of tree estimators. Thus, we chose model sizes that are sufficient to get a better accuracy on classification datasets (Table 4 and Table 5).

Table 4. Epsilon (100K rows)

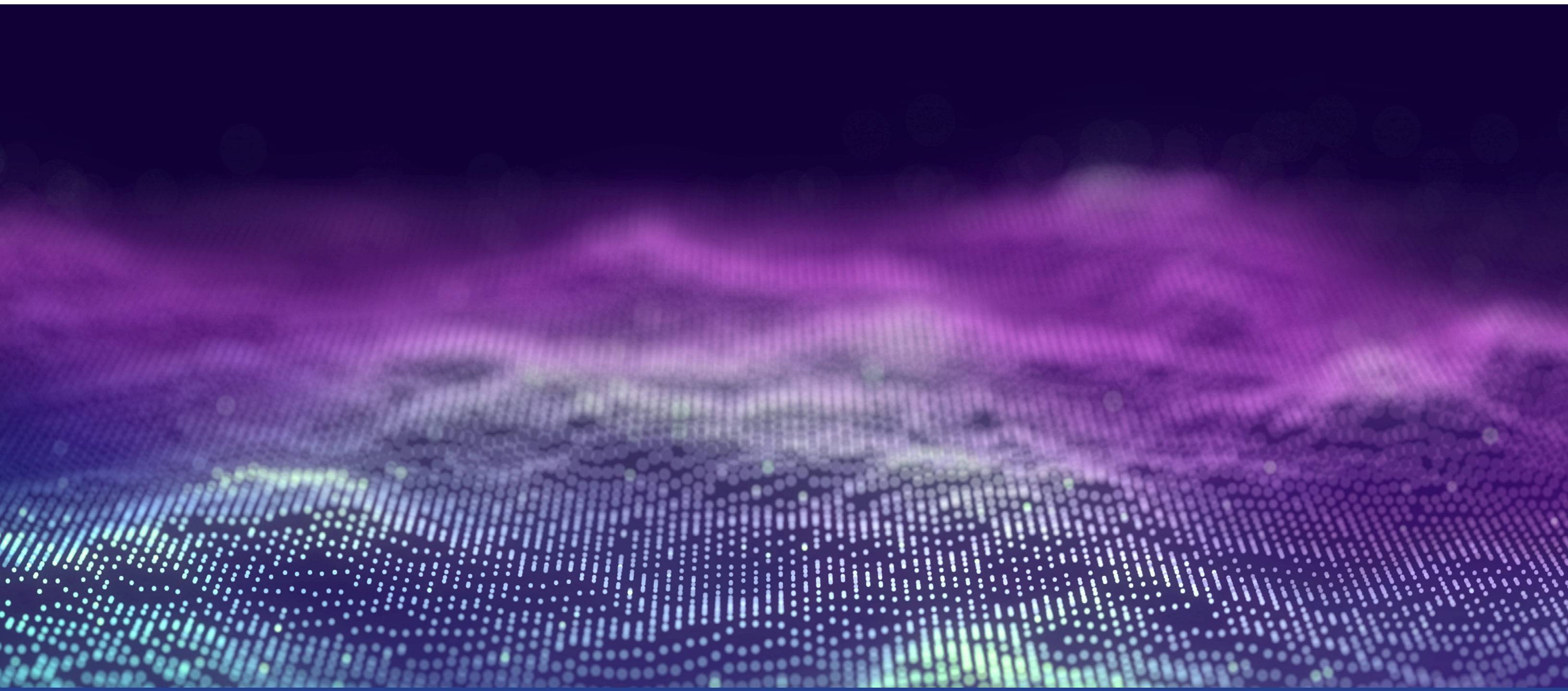
No. of Trees/ Max. Depth	100-4	100-8	100-12	...	900-12	1000-4	1000-6	1000-8	1000-12
Classifi-cation Accuracy	0.81504	0.83103	0.8258	...	0.8751	0.87624	0.87431	0.8726	0.86559

Table 5. Bosch (100K rows)

No. of Trees/ Max. Depth	100-4	100-8	100-12	200-4	200-8	...	1000-4	1000-8	1000-12
Classifi-cation Accuracy	0.99418	0.99467	0.99469	0.99421	0.99451	...	0.99441	0.99429	0.99387

Inference accuracy on a selected model (Epsilon “1000-4”) is approximately the same with Intel DAAL for FIL and XGBoost CPU/GPU: 0.874400. On another selected model (Bosch “100-12”) accuracy is also approximately the same with Intel DAAL for FIL and XGBoost CPU/GPU: 0.99444.

For regression inference on the Mortgage dataset, we used the parameters published by NVIDIA for their RAPIDS **benchmarks** (the trained model has 100 tree estimators with maximum depth equal to 8).



K-MEANS ACCELERATION WITH 2ND GENERATION INTEL® XEON® SCALABLE PROCESSORS

Intel's Hardware and Software Stack for Big Data

Alexander Andreev, Machine Learning Engineer, and Egor Smirnov, Software Engineering Manager, Intel Corporation

The amount of data humans produce every day is growing exponentially—and so is the need for high-performance, scalable algorithms to process and extract benefit from all this data. Intel puts a lot of effort into building effective data analytics tools for our platforms by optimizing not only hardware, but also software. In recent years, Intel has built a powerful data analytics and machine learning (ML) software and hardware stack that includes both optimized frameworks[1] and libraries.[2]

From the hardware side, high performance and scalability are provided by **Intel® Xeon® Scalable processors**[3] (the 2nd generation also contains **Intel® Deep Learning Boost [Intel® DL Boost] technology**], **Intel® Nervana™ Neural Network Processors [Intel® Nervana™ NNP]**, **Intel® FPGAs**, and **Intel® Movidius Neural Compute Stick**[4]).

Intel-Optimized scikit-learn* and Intel® Data Analytics Acceleration Library (Intel® DAAL)

One of Intel's optimized frameworks for classic ML algorithms, scikit-learn*[5], is part of the **Intel® Distribution for Python* (IDP)**.[6] The IDP scikit-learn uses the **Intel® DAAL**[7] library underneath to obtain high performance on Intel® architectures. Intel DAAL is an open-source[8] data analytics library optimized for Intel® architectures ranging from mobile (**Intel® Atom® processors**) to data centers (**Intel® Xeon® processors**). Intel DAAL provides C++, Java*, and Python APIs, as well as newly introduced **Data Parallel C++ (DPC++)**, which is part of **oneAPI**,[9] a unified programming model for effective development on different architectures (CPU, GPU, and others).

K-means is one of the accelerated ML algorithms in IDP scikit-learn.

The K-means Algorithm

K-means, a popular clustering algorithm, is used in astronomy, medicine, market segmentation, and many other areas. It's one of the accelerated ML algorithms implemented in Intel DAAL. k-means is both simple and powerful. Its objective is to split a set of N observations into K clusters. This is achieved by minimizing inertia (i.e., the sum of squared Euclidian distances from observations to the cluster centers [centroids]). The algorithm is iterative, with two steps in each iteration:

1. **For each observation**, compute the distance from it to each centroid, and then reassign each observation to the cluster with the nearest centroid.
2. **For each cluster**, compute the centroid as the mean of observations assigned to this cluster.

Repeat these steps until one of the following happens:

1. **The number of iterations** exceeds its predefined maximum.
2. **The algorithm converges** (i.e., the difference between two consecutive inertias is less than a predefined threshold).

Different initialization methods are used to get initial centroids for the first iteration. It can select random observations as initial centroids, or use more complex methods such as k-means++.[10]

Intel DAAL versus RAPIDS cuML* on Distributed k-means

To show how well Intel DAAL accelerates k-means, we compared it to RAPIDS cuML, which claims to be 90x faster than CPU-based implementations.[11] We also compared cuML to scikit-learn from IDP, which uses Intel DAAL underneath.

From Amazon Web Services Elastic Compute Cloud*[12] (AWS EC2*), we used the following instances (nodes) to measure performance:

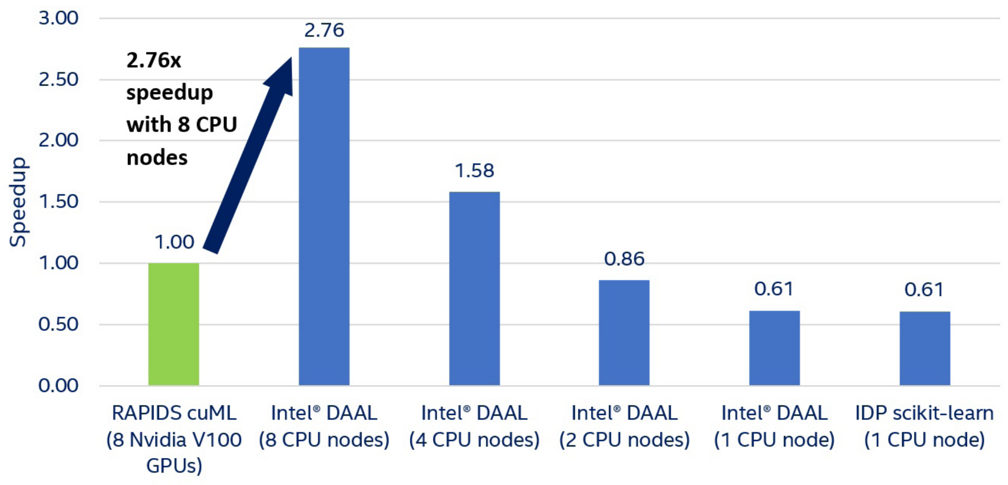
- **Multiple (up to 8) AWS EC2 `c5.24xlarge` instances** with 2nd Generation Intel Xeon Scalable processors
- **One AWS EC2 `p3dn.24xlarge` instance** with eight Nvidia V100* GPUs

We chose the maximum number of `c5.24xlarge` instances to be eight because the cost per hour of eight `c5.24xlarge` instances is approximately the same as one `p3dn.24xlarge` instance.

We used a synthetic dataset with 200 million observations, 50 columns, and 10 clusters, which is as much as the V100's memory can store. A similar dataset with 300 million observations instead of 200 caused a RAPIDS memory error (`RMM_ERROR_OUT_OF_MEMORY`). CPU-based systems can typically process much larger datasets. (The code for dataset generation is shown in the Code Examples section at the end of this article.)

We chose the same initialization method in all measured cases for an apples-to-apples comparison, and we chose the float32 datatype since it gives sufficient accuracy and fits in memory.

Figure 1 and **Table 1** show that starting from four Intel Xeon Scalable processor nodes, Intel DAAL outperforms cuML on eight V100 GPUs. Even one node with two Intel Xeon Scalable processors is only 40% slower than eight V100 GPUs. Also, it can hold and process the whole dataset, while fewer than eight V100s can't. IDP scikit-learn and Intel DAAL on one CPU node show approximately the same results: the difference in training time is trivial, and due to Intel DAAL function call overhead.



1 **Speedup of k-means training with Intel DAAL**

Table 1. K-means training time (in seconds) and speedup: comparison of IDP scikit-learn, Intel® DAAL, and RAPIDS cuML

Library	Training Time, s	Speedup
IDP scikit-learn (1 CPU node)	20.40	0.61
Intel® DAAL (1 CPU node)	20.21	0.61
Intel DAAL (2 CPU nodes)	14.41	0.86
Intel DAAL (4 CPU nodes)	7.84	1.58
Intel DAAL (8 CPU nodes)	4.49	2.76
RAPIDS cuML (8 Nvidia V100 GPUs)	12.41	1.00

The k-means algorithm converges to the same result for all measured cases. Inertia is 4×10^8.

The IDP scikit-learn and Intel DAAL examples are available in the Code Examples section.

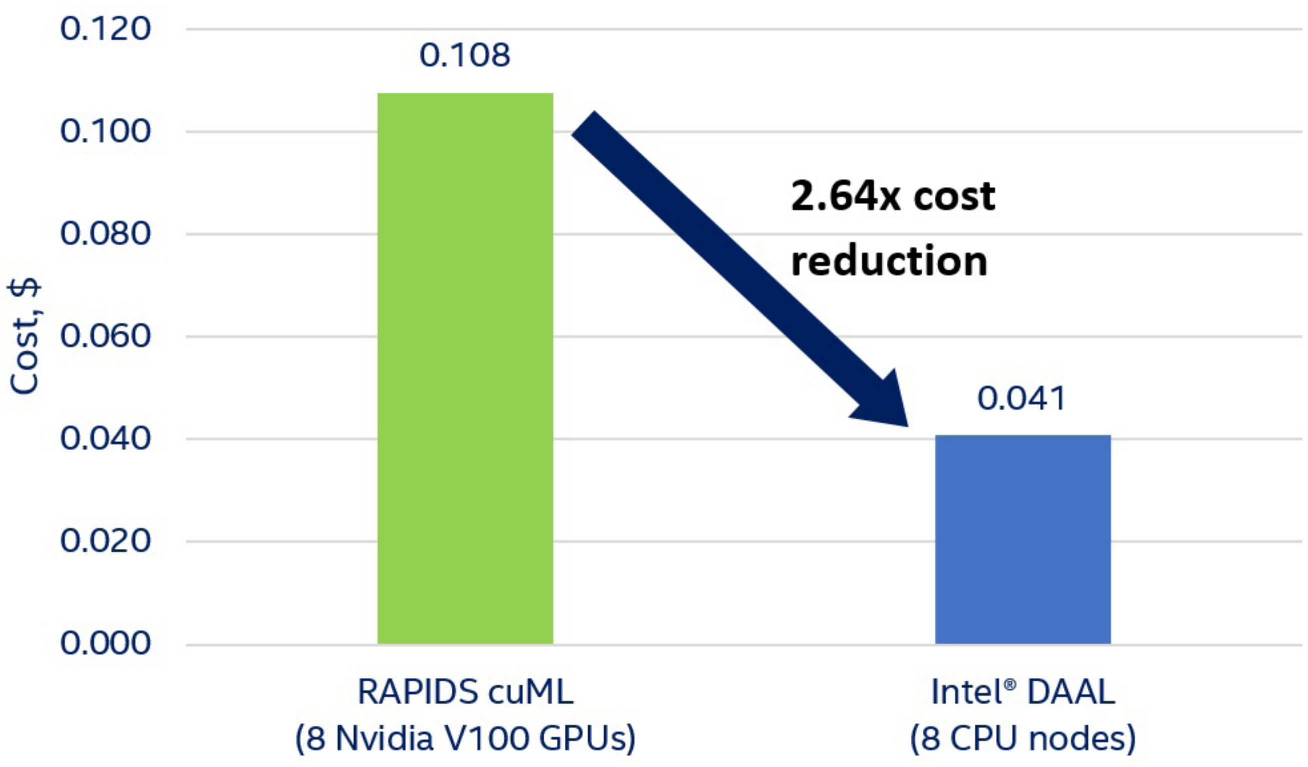
Moreover, we calculated the k-means training cost as the cost of instances on AWS EC2 multiplied by the training time for eight Intel Xeon processor nodes and eight V100 GPUs, respectively.

$$\text{k-means training cost (\$)} = \frac{\text{Cost of AWS EC2 instances (\$/hour)}}{\text{k-means training time (hours)}}$$

Figure 2 shows that using Intel DAAL with eight CPU nodes results in up to 2.64x reduction in k-means training cost.

Faster and Cheaper

On AWS EC2, distributed k-means computations are 2.76x faster and 2.64x cheaper with Intel DAAL on eight Intel Xeon processor nodes than with RAPIDS cuML on eight V100s. Moreover, Intel Xeon processor-based instances can process larger datasets: data that was easily processed on one Intel Xeon processor-based node causes “out of memory” errors on eight V100s. With Intel Optane,[13] memory capacity increases to 4.5 TB per socket (9 TB per two-socket instance), while an NVIDIA DGX-2 has only 512 GB of GPU memory.



2 **k-means training cost:[19] RAPIDS cuML on 8 GPUs and Intel DAAL on eight CPU nodes. AWS EC2 (N. Virginia) instance prices: `c5.24xlarge` (Intel DAAL) - $4.08 per hour ($32.64 per hour for eight nodes), `p3dn.24xlarge` (RAPIDS cuML) - $31.212 per hour.**

With the oneAPI programming model,[9] Intel delivers high performance, not only on the CPU, but also on its coming architectures (discrete GPU, FPGA, and other accelerators). oneAPI is delivering unified and open programming experience to developers on the architecture of their choice without compromising performance. Intel DAAL is a part of the beta Intel oneAPI products. You can try it in the Intel® DevCloud development sandbox.[14]

Intel DAAL k-means Implementation and Optimization Details

The k-means algorithm is based on computation of distances, since it's used in cluster assignments and objective function (inertia) calculation. The distance in d-dimensional Euclidean space between an

$$D = \sqrt{\sum_{i=0}^{d}(s_i - c_i)^2}$$

This is equal mathematically to:

$$= \sqrt{\sum_{i=0}^{d} (s_i^2 - 2s_i c_i + c_i^2)}$$

Thus, the squared distance is:

$$D^2 = \sum_{i=0}^{d} (s_i^2 - 2s_i c_i + c_i^2)$$

To calculate squared distances, the k-means implementation in Intel DAAL splits observations into blocks of fixed size and then processes them in parallel using **Threading Building Blocks**.[15]

Component $\sum_{i=0}^{d} 2s_i c_i$ presented as an element m_{jn} of matrix M = 2 S × C, where j is an index of an observation in a block, n is an index of a centroid, and S and C are matrices of a block of observations and centroids, respectively. To calculate this matrix of distance components, Intel DAAL uses matrix multiplication from **Intel MKL**.[16]

The first squared distance component, $\sum_{i=0}^{d} s_i^2$ is constant and can be calculated only once for each observation, while two others ($\sum_{i=0}^{d} 2s_i c_i$ and $\sum_{i=0}^{d} c_i^2$) should be recalculated at each iteration.

At all k-means computation stages, Intel DAAL uses vector instructions from **Intel® Advanced Vector Extensions 512 (Intel® AVX-512)**[17] in 2nd-generation Intel Xeon Scalable processors when computing matrix multiplication, centroid assignments, and centroid recalculation.

Using advanced software optimization techniques and enabling hardware features allows Intel DAAL to deliver high performance k-means clustering.

IDP scikit-learn and daal4py Installation

The best way to install scikit-learn from IDP or daal4py (the Python interface for Intel DAAL)[18] is by creating a new conda environment:

```
conda create -n idp -c intel daal4py scikit-learn pandas
```

Code Examples

Dataset generation with scikit-learn:

```
from sklearn.datasets.samples_generator import make_blobs
import numpy as np

x, y = make_blobs(n_samples=2*10**8, n_features=50, centers=10,
    cluster_std=0.2, center_box=(-10.0, 10.0), random_state=777)
np.savetxt("kmeans_data.csv”, x, fmt="%f", delimiter=",")
```

Running k-means with scikit-learn from IDP:

```
import numpy as np
import pandas as pd
import sklearn as skl

# set config to prevent long finiteness check of input data
skl.set_config(assume_finite=True)

# fast csv reading with pandas
train_data = pd.read_csv("kmeans_data.csv”, dtype=np.float32)

alg = skl.cluster.KMeans(n_clusters=10, init="random",
    tol=0, algorithm="full", max_iter=50)
alg.fit(train_data)
```

Running k-means with daal4py:

```
import numpy as np
import pandas as pd
import daal4py as d4p

d4p.daalinit()
# fast csv reading with pandas
data = pd.read_csv("local_kmeans_data.csv", header=None, dtype=np.float32)

centroids = d4p.kmeans_init(nClusters=10, fptype="float",
    method="randomDense", distributed=True)
alg = d4p.kmeans(nClusters=10, maxIterations=50, fptype="float",
    accuracyThreshold=0, assignFlag=False, distributed=True)
result = alg.compute(data, centroids)
```

k-means Training Configuration

- **CPU configuration:** c5.24xlarge AWS EC2 instances; 2nd generation Intel Xeon Scalable processors, two sockets, 24 cores per socket, HT on, Turbo on, 192 GB RAM (12 slots/16GB/2933 MHz), BIOS: 1.0 Amazon EC2* (ucode: 0x500002c), OS: Ubuntu* 18.04.2 LTS.
- **GPU configuration:** p3dn.24xlarge AWS EC2 instance; Intel® Xeon® Platinum 8175M processor, two sockets, 24 cores per socket, HT on, Turbo on, 768 GB RAM, 8 Tesla V100-SXM2-32G* GPUs with 32 GB GPU memory each, BIOS 1.0 Amazon EC2* (ucode: 0x2000065), OS: Ubuntu 18.04.2 LTS.
- **Software:** Python* 3.6, numpy* 1.16.4, scikit-learn 0.21.3, daal4py 2019.5, Intel DAAL 2019.5, Intel® MPI 2019.5, RAPIDS cuML 0.10, RAPIDS cuDF* 0.10, CUDA* 10.1, Nvidia* GPU driver 418.87.01, dask* 2.6.0.
- **Algorithm parameters:** Single-precision (float32), number of iterations = 50, threshold = 0, random initial centroids.

References

1. **Intel-optimized frameworks**
2. **Intel-optimized libraries**
3. **Intel Xeon Scalable processors**
4. **Intel hardware stack for AI**
5. **scikit-learn**
6. **Intel Distribution for Python**
7. **Intel DAAL**
8. **Intel DAAL on GitHub**
9. **Fact sheet: oneAPI**
10. **k-means++**
11. **RAPIDS 0.9: A Model Built to Scale, section "A Model Built to Scale: cuML"**
12. **Amazon Web Services Elastic Compute Cloud**
13. **Intel® Optane**
14. **Intel® DevCloud**
15. **Intel® Threading Building Blocks**
16. **Intel® MKL**
17. **Intel® Advanced Vector Extensions 512**
18. **daal4py: A Convenient Python API to Intel® DAAL**
19. **AWS EC2 Pricing as of 12/02/2019. URL**

HOW'D THEY DO THAT?
Developers worldwide have upped the ante for application performance, scalability, and portability with Intel® Software Development Tools.
And they're sharing their stories to help you do the same.
EXPLORE >
intel®
Software

MEASURING GRAPH ANALYTICS PERFORMANCE

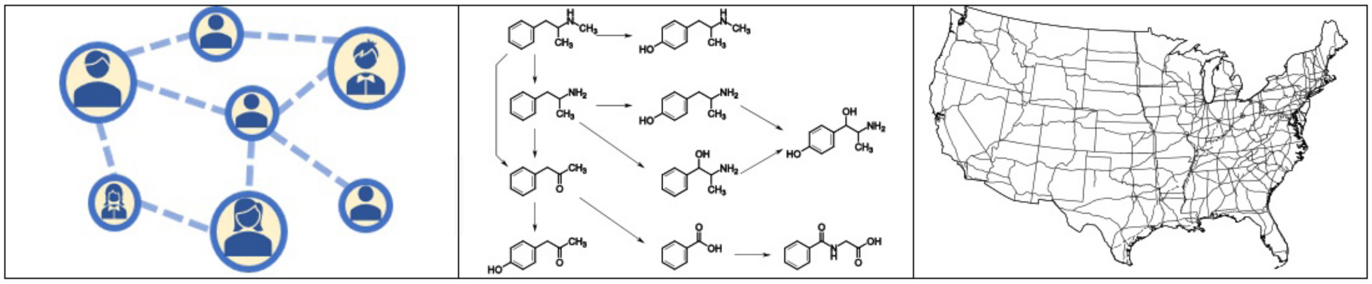
What Is Graph Analytics? And Why Does It Matter?

Henry A. Gabb, Senior Principal Engineer, Intel Corporation, and Editor, The Parallel Universe

A graph is a good way to represent a set of objects and the relations between them (**Figure 1**). Graph analytics is the set of techniques to extract information from connections between entities.

For example, graph analytics can be used to:

- **Get recommendations** among friends in a social network
- **Find cut points** in a communication network or electrical grid
- **Determine drug effects** on a biochemical pathway
- **Detect robocalls** in a telecommunications network
- **Find optimal routes** between locations on a map



1 **Graphs are everywhere**

Graph analytics has been getting a lot of attention lately, possibly because Gartner listed it among the top 10 data and analytics technology trends for 2019:

> *"The application of graph processing and graph DBMSs will grow at 100 percent annually through 2022 to continuously accelerate data preparation and enable more complex adaptive data science."* (Source: **Gartner Identifies Top 10 Data and Analytics Technology Trends for 2019**)

Intel has a long history of leadership in graph analysis. For example, Intel coauthored the **GraphBLAS specification** to formulate graph problems such as sparse linear algebra. Though the **GraphBLAS API** was just published in 2017, the initial proposal and manifesto were published over 10 years ago. Today, the same industry and academic partnership is coauthoring the forthcoming **LAGraph specification** for a library of graph algorithms. Intel was also selected by DARPA to develop a new processor to handle large graph datasets (see "**DARPA Taps Intel for Graph Analytics Chip Project**"). We'll continue to innovate and push the graph analytics envelope.

Benchmarking Graph Analytics Performance

Graph analytics is a large and varied landscape. Even the simple examples in **Figure 1** show differing characteristics. For example, some networks are highly connected; some are sparser. A network of webpages exhibits different connectivity than a network of Twitter* users, where some users have millions of followers while most have only a few. Consequently, no single combination of graph algorithm, graph topology, or graph size can adequately represent the entire landscape.

Therefore, we use the **GAP Benchmark Suite** from the University of California, Berkeley, to measure graph analytics performance. GAP specifies six widely-used algorithms (**Table 1**) and five small to medium-sized graphs (**Table 2**). Each graph has different characteristics, which is important because optimizations that work well for one graph topology may not work well for others. For example, the Road* graph is relatively small, but its high diameter can cause problems for some algorithms. Consequently, the 30 GAP data points provide good coverage of the graph analytics landscape.

Table 1. GAP measures the performance of six common graph analytics algorithms.

Graph Algorithm	Benchmark Definition
Breadth-first search (BFS)	The average time of 64 traversals from 64 random starting vertices
Single-source shortest path (SSSP)	The average time to compute the shortest paths between 64 random source vertices and every other reachable vertex
Connected components (CC)	The average time of 16 CC computations
PageRank (PR)	The average time of 16 PR computations (tolerance = 0.0001, maximum iterations = 1,000)
Triangle counting (TC)	The average time of three TC computations
Betweenness centrality (BC)	The average time of 16 trials, each with BC computations for four random vertices

Table 2. GAP uses five graphs of varying size and topology to give a more complete picture of graph analytics performance.

Graph	Vertices (Millions)	Edges (Millions)	Type	Topological Characteristics
Twitter	61.6	1,468.4	Real	Skewed degree distribution
Web	50.6	1,949.4	Real	High average degree
Road	23.9	58.3	Real	High diameter
Kron	134.2	2,111.6	Synthetic	Scale-free Kronecker graph
Urand	134.2	2,147.4	Synthetic	Non-scale-free Erdős-Réyni graph

GAP also has the advantages of being a clearly-defined, objective, off-the-shelf benchmark. It doesn't require special hardware or software configurations, so it's easy to run.

Here are the steps I took to run the benchmark:

1. Download the **GAP package**.
2. Run ‘`make`’ in the `gapbs-master` subdirectory to build the reference implementations for the six graph analytics kernels. For now, I'm just using the default GAP build parameters (the GNU* C++ compiler with the `-std=c++11 -O3 -Wall -fopenmp` options). My system had the GNU v7.4.0 compiler installed.
3. In the same directory, run ‘`make -f benchmark/bench.mk bench graphs`’ to download or generate the five benchmark graphs and convert them to more efficient input formats.

4. The GAP reference implementations are parallelized with OpenMP*, so it's important to set the number of threads. Otherwise, GAP will use all available cores, even if this doesn't give the best parallel efficiency. The `'export OMP_NUM_THREADS=32'` command sets the number of OpenMP threads to 32, for example.
5. Finally, run `'make -f benchmark/bench.mk bench run'` to launch the benchmarks and generate results (**Table 3**).

Table 3. Compute times (in seconds, lower is better) for GAP running on a two-socket Intel® Xeon® processor-based system[1]. GAP was run with 1, 8, 16, 24, 32, 48, 64, and 96 threads. Best performance is shown for each test.

Graph	Graph Algorithm					
	BFS	SSSP	CC	PR	TC	BC
Twitter	0.2	1.8	0.2	7.4	39.0	7.5
Web	0.3	0.8	0.2	4.4	14.9	2.5
Road	0.3	1.0	0.04	1.0	0.02	2.4
Kron	0.3	3.6	0.6	10.6	183.3	17.3
Urand	0.4	5.6	1.1	16.0	12.2	24.8

I tried to generate NVIDIA V100* comparative data, but ran into several technical barriers:

1. Only one of the GAP graphs (Road) fits in the memory of a single V100.
2. Only one of the GAP algorithms (PR) in RAPIDS cuGraph* can use the aggregate memory of multiple V100s.
3. The current version of cuGraph does not provide a BC implementation.
4. The cuGraph APIs and documentation do not expose certain implementation details or algorithm parameters. For example, the multi-V100 PR implementation in cuGraph does not provide a convergence tolerance parameter, which makes an apples-to-apples comparison to the GAP results difficult.

Consequently, it's only possible to run nine of the 30 GAP tests (**Table 4**). As you can see, where GAP can run on both architectures, the Intel® Xeon® processors outperformed the V100 on most tests, even when the graph is small enough to fit in GPU memory (i.e., Road) or when the algorithm can use multiple GPUs (i.e., PageRank). Also, TCO clearly favors Intel Xeon processors for graph analytics (**Table 5**).

Table 4. Compute times (in seconds, lower is better) for cuGraph running on an AWS EC2 p3.16xlarge instance (eight V100 GPUs connected via NVLink). All Road tests used a single V100. Twitter and Web tests used eight V100s. Note that the multi-V100 PR in cuGraph does not provide a convergence tolerance parameter so the default parameters were used for these tests. Kron and Urand tests failed with a "`thrust::system::system_error.`" DNR = Does not run because of insufficient memory. NA = Not available in cuGraph v0.9.0.

Graph	Graph Algorithm					
	BFS	SSSP	CC	PR	TC	BC
Twitter	DNR	DNR	DNR	14.8	DNR	NA
Web	DNR	DNR	DNR	26.8	DNR	NA
Road	0.5	25.7	94.1	0.2	0.01	NA
Kron	DNR	DNR	DNR	Failed	DNR	NA
Urand	DNR	DNR	DNR	Failed	DNR	NA

Table 5. Relative TCO of the Intel Xeon processor and V100 benchmark systems. Note that the AWS price comparison is only an approximation because no EC2 instances exactly match the Intel Xeon processor-based benchmarking system, but the m4.16xlarge instance is similar.

Metric	Intel® Xeon® Processor-Based System	V100-Based System
Processor	2 x Intel® Xeon® Gold 6252 processor	8 x NVIDIA Tesla* V100 SXM2 (16 GB HBM2)
Thermal Design Power	2 x 150W = 300W	8 x 300W = 2,400W
Price	2 x $3,655 = $7,310	8 x $10,664 = $85,312
AWS EC2 Instance Price	m4.16xlarge: $3.20/hour	p3.16xlarge: $24.48/hour

Comprehensive, Objective, and Reproducible

I can probably improve the GAP results on the Intel Xeon processor platform (**Table 3**) with a few simple changes like using the Intel® compiler and aggressive optimization and vectorization, tweaking the OpenMP `OMP_PROC_BIND and OMP_PLACES` environment variables, experimenting with the `numactl` utility, adjusting page sizes, etc.—but that's not the purpose of this article. My goal is to show how easy it is to get comprehensive, objective, and reproducible graph analytics performance data without obfuscation or resorting to benchmarking tricks.

References

1. Processor: Intel® Xeon® Gold 6252 (2.1 GHz, 24 cores), HyperThreading enabled (48 virtual cores per socket); Memory: 384 GB Micron DDR4-2666; Operating system: Ubuntu Linux* release 4.15.0-29, kernel 31.x86_64; Software: GAP Benchmark Suite (downloaded and run September 2019). Processor: NVIDIA Tesla V100*; Memory: 16 GB; Operating system: Ubuntu Linux release 4.15.0-1047-aws, kernel 49.x86_64; Software: RAPIDS v0.9.0 (downloaded and run September 2019).
2. Source: https://www.microway.com/hpc-tech-tips/nvidia-tesla-v100-price-analysis/ (accessed September 2019) and https://ark.intel.com/content/www/us/en/ark/products/192447/intel-xeon-gold-6252-processor-35-75m-cache-2-10-ghz.html (accessed September 2019)
3. Source: https://aws.amazon.com/ec2/pricing/on-demand/ (accessed September 2019)
4. See "Boosting the Performance of Graph Analytics Workloads" if you're interested in tuning GAP.

Software

THE PARALLEL UNIVERSE